



TU Wien

Business Informatics Group
Institut für Softwaretechnik und Interaktive Systeme

Model Driven Architecture in der Praxis
Evaluierung aktueller Entwicklungs-werkzeuge und Fallstudie

Diplomarbeit zur Erlangung des akademischen Grades eines Magister der Sozial- und Wirtschaftswissenschaften

eingereicht bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
mitbetreuender Assistent: Dipl.-Ing. Dr. Gerhard Kramler

Manuel Wimmer

Wien, 8. März 2005

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 8. März 2005 Manuel Wimmer

Danksagung

Ich möchte mich bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel und Dipl.-Ing. Dr. Gerhard Kramler für die Anregung zu dieser Diplomarbeit und für die Unterstützung bei der Erstellung der Arbeit bedanken. Weiters möchte ich mich ganz besonders bei meiner Mutter Ingrid Wimmer bedanken, die mir durch ihre finanzielle und moralische Unterstützung mein Studium und dadurch auch diese Diplomarbeit ermöglichte. Außerdem bedanke ich mich bei den KorrekturleserInnen: Michael Edinger, Christian Schlosser und Sabine Wimmer.

Kurzfassung

Model Driven Architecture (MDA) wird von der Object Management Group (OMG) stark vorangetrieben und hält Einzug in die Softwareentwicklung. MDA wird als nächster Schritt des Formalisierungs- und Abstraktionsprozesses von Software gesehen, etwa wie die Ablöse von Assembler durch Hochsprachen. Schenkt man den vielen Versprechungen der OMG Glauben, so lassen sich aus technologieunabhängigen Modellen problemlos Anwendungen für verschiedenste Plattformen wie J2EE, CORBA oder .NET automatisch generieren.
Dabei gibt die OMG nur eine theoretische Architektur und Spezifikation von MDA vor, die Implementierungen von Werkzeugen bzw. Transformationsregeln wird an die Softwareindustrie abgegeben. Diese soll in nächster Zeit für Automatisierungstools sorgen, die die Modelltransformationen und weitere Hilfsmittel für den erfolgreichen Einsatz von MDA unterstützen.

In der Praxis wurden schon mehrere Projekte mittels MDA auf Basis der bereits vorhandenen Werkzeuge verwirklicht. Die Vision dabei ist, dass Softwareanwendungen automatisch generiert werden. Hier ist aber zu klären, ob die Tools wirklich Anspruch auf MDA haben oder ob es sich „nur“ um UML Tools handelt, die Codesegmente automatisch aus grafischen Modellen ableiten können.

Die Diplomarbeit beschreibt ausführlich die theoretischen Konzepte von MDA sowie die praktische Umsetzung von MDA anhand eines Ausschnitts des CALENDARIUM-Projektes. Die theoretischen Konzepte wie plattformunabhängiges Modell, plattformabhängiges Modell, Modelltransformationen und Metamodellierung werden ausführlich erklärt und diskutiert. Dabei werden einerseits Probleme der Transformation der verschiedenen Modelle (vom Business Modell bis hin zum Code) aufgezeigt, sowie die Auswirkungen auf den Softwareentwicklungsprozess untersucht. Weiters wird auf die manuelle Erstellung von Transformationsregeln eingegangen.
Anhand eines Kriterienkatalogs werden die Anforderungen an MDA-Werkzeuge diskutiert und existierende Werkzeuge untersucht. Weiters wird anhand des CALENDARIUMS ein Beispiel MDA-Projekt auf Basis einer relationalen Daten-

bank, Servlets, JSPs und EJBs durchgeführt. Dabei wird mit Hilfe von Tools die Erstellung eines lauffähigen Programms aus einem plattformunabhängigen Modell durchgeführt.

Inwieweit die MDA-Werkzeuge diese Aufgabe automatisch erledigen können, oder an welchen Stellen die ProgrammiererIn selber noch Hand anlegen muss, wird durch dieses Beispielprojekt dargestellt. Weiters wird durch die Tool-Evaluierung der aktuelle Stand der Technik von MDA-Werkzeugen aufgezeigt. Dabei werden einerseits Bereiche von MDA identifiziert, die durch aktuelle Werkzeuge bereits umgesetzt wurden, und andererseits aufgezeigt, welche Bereiche unzureichend abgedeckt werden und noch weitere Forschungsaktivitäten benötigen.

Abstract

Model Driven Architecture (MDA) is strongly propagated by the Object Management Group (OMG) and is being introduced into software development practice. MDA is seen as the next step in the formalizing and abstraction process of software, like the replacement of assembler by high-level languages. If we believe in the promises of the OMG, then full working applications for different software platforms (e.g. J2EE, .NET, CORBA) can be automatically generated from platform independent models.
The OMG defines a theoretical architecture and specification of MDA, but the implementations of tools and transformation rules are transferred to the software industry. Tool manufacturers should develop MDA tools, which support model transformations and further aid for the successful employment of MDA in practical software development.

In practice several projects were already realized with MDA support on basis of already existing tools. The vision of the OMG is that applications are automatically generated from models. Here it has to be clarified whether the tools really implement MDA concepts or whether they are no more than UML tools, which can derive code segments automatically from graphical models.

This diploma thesis describes in detail the theoretical concepts of MDA as well as the practical MDA aspects on the basis of the CALENDARIUM project. The theoretical concepts like platform independent models, platform dependent models, model transformations and metamodels are explained. On the one hand problems of the transformation of the different models are pointed out and on the other hand effects on the software development process are discussed. Further the manual definition of transformation rules is explained.
On the basis of a list of criteria requirements for MDA tools are collected and existing tools are examined. On the basis of the CALENDARIUM project an example MDA project, which uses a relational database, Servlets, JSPs and EJBs, is constructed. In this case study a MDA tool is used for the production of an executable application from a platform independent model.

This case study demonstrates, to what extent MDA tools can perform the task automatically, or on which places the programmer still has to do manual programming. Further the current state of the art of MDA tools is pointed out by the tool evaluation. On the one hand areas are identified, which are already supported by current MDA tools, and on the other hand it is shown, which areas are covered insufficiently and still need further research activities.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation für MDA

Heutzutage ist die Softwareentwicklung mit immer größer und komplexer werdenden Systemen konfrontiert, man denke an Online-Bestellsysteme wie z.B. Amazon oder an den Einsatz von mobilen Geräten, wie z.B. PDAs oder Mobiltelefonen. Mit objektorientierter Programmierung und verbesserten Softwareentwicklungsprozessen versuchen die SoftwareentwicklerInnen und SoftwarearchitektInnen den hohen Programmieraufwand in den Griff zu bekommen. Aber dennoch bleibt die Softwareentwicklung trotz Fortschritten noch immer äußerst arbeitsintensiv und dadurch auch teuer, da ein Großteil der Arbeitsschritte nicht automatisiert abläuft, sondern von den ProgrammiererInnen händisch durchgeführt werden muss. Mit jeder Einführung von neuen Softwaretechnologien entstehen Modifikationen an bestehenden Softwaresystemen. Bei gravierenden Umstellungen der Technologien bleibt eine Re-Implementierung der Softwaresysteme nicht aus und eine Umschulung des Personals stellt ein Muss für ein erfolgreiches Projekt dar. Weiters benötigt ein nicht triviales Softwaresystem für die Erfüllung seiner Aufgaben mehrere verschiedene Technologien und kommuniziert über Netzwerke, wie z.B. das Internet, mit anderen Systemen. Um eine effiziente Nutzung einer Applikation zu erreichen, wird eine Vielzahl der heute entwickelten Applikationen als verteilte Systeme (d.h. über mehrere Rechner verteilt ablaufend) entworfen.

Die erwähnten Entwicklungen fordern neue Entwicklungsmethoden, um die Erstellung von Software zu erleichtern bzw. zu automatisieren. Diese Erkenntnis ist bei weitem keine neue, denn Dijkstra [Dijk72] bemerkte bereits im Jahr 1972:

„The art of programming is the art of organising complexity".

Trotz Verbesserungen in Softwareentwicklungsprozessen und Programmiertechniken, befindet sich die Produktivität im Bereich Softwareentwicklung immer noch auf einem zu niedrigen Niveau. [Mell04] erklärt dafür folgende Gründe, welche auf Mängel in Programmierkonzepten und Programmiersprachen basieren:

1. Code besitzt eine zu niedrige Abstraktionsebene
2. Wiederverwendung basiert auf einer zu feinen Form
3. Programmspezifikationen sind stark mit ihrer Infrastruktur verflochten
4. „Business to Software Mappings" werden nicht archiviert

Diese vier Hauptprobleme der Softwareentwicklung werden in den nächsten Abschnitten genauer erläutert. Dabei werden potentielle Lösungsmöglichkeiten durch den Einsatz der modellgetriebenen Softwareentwicklung angeführt und ausführlich beschrieben.

Code besitzt eine zu niedrige Abstraktionsebene

In Programmcode, wie z.B. Java oder C#, werden viele plattformspezifische Details und Angaben über Datenstrukturen (wie z.B. Hashtables oder Arrays) oder Algorithmen (wie z.B. Such- oder Sortieralgorithmen) implementiert. Die eigentliche Problemstellung (Domänenwissen) ist nicht unmittelbar erkennbar, da sie über viele Textdateien fragmentiert vorliegt und mit technologischen Details vermischt ist.

Dieses Problem ruft nach einer neuen, höheren Abstraktionsebene für Softwarespezifikationen, um mit immer komplexer werdenden und einer stark wachsenden Anzahl von verschiedenartigen Softwaresystemen (wie z.B. mobile Geräte, *Embedded Systems*) fertig zu werden. Die Antwort, welche von der Object Management Group (OMG) propagiert wird, heißt *Model Driven Architecture* (*MDA*).

Dieser Softwareentwicklungsansatz ist ein weiterer evolutionärer Schritt in der Geschichte der Softwareentwicklung. Im Mittelpunkt dieses Ansatzes stehen Modelle, die Softwaresysteme auf unterschiedlichen Ebenen beschreiben. Die Aufgabe der

Modelle ist nicht nur die Kommunikation zwischen EntwicklerInnen, ArchitektInnen und KundInnen oder die Dokumentation des Systems, sondern die Modelle werden für die Gewinnung neuer, verfeinerter Modelle und weiters für die Generierung von lauffähigem Code verwendet. Im MDA-Paradigma werden primär zwei Arten von Modellen unterschieden:

- *Platform Independent Model* (*PIM*): enthält nur den Problem- bzw. Anwendungsbereich jedoch keine Details über Software- bzw. Hardwaretechnologien

- *Platform Specific Model* (*PSM*): enthält zusätzlich zum PIM Details über die verwendeten Software- bzw. Hardwaretechnologien

Im Idealfall läuft der Übergang von einem plattformunabhängigen Modell auf ein plattformspezifisches Modell vollständig automatisiert ab und weiters wird aus dem plattformspezifischen Modell automatisch lauffähiger Code transformiert. Abbildung 1.1 beschreibt den Kerngedanken des MDA-Paradigmas.

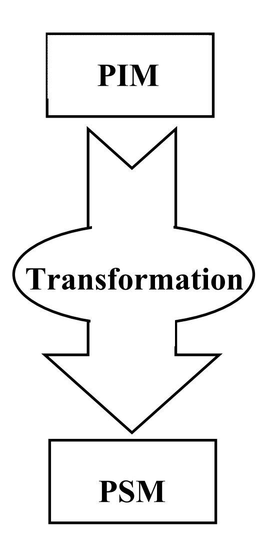


Abbildung 1.1 Modell Transformation (PIM -> PSM)

Die automatische Erstellung von Software aus plattformunabhängigen Modellen wird mit der Einführung einer weiteren Abstraktionsebene – das PIM stellt eine neuartige Abstraktionsebene dar – in der Softwareentwicklung ermöglicht. In der Vergangenheit wurden bereits verschiedene Abstraktionsebenen in den Programmiersprachen eingeführt. Am Anfang der Geschichte der Softwareentwicklung wurden Programme als Maschinencode (Folge von Nullen und Einsen) geschrieben und weiters auch in dieser Form gespeichert. Da noch viele Details der verwendeten Hardware im Code berücksichtigt wurden, war ein Programm nur auf der dafür vorgese-

henen Maschine lauffähig. Die erste Abstraktion von Maschineninstruktionen war die Einführung von Assemblersprachen. Damit war es möglich, Programme für eine bestimmte Hardwareplattform als Sammlung von *Mnemonics* zu entwickeln. Mit der Einführung von Hochsprachen, wie z.B. Fortran und C, wurde die nächste Abstraktionsstufe erreicht. Vorerst herrschte Ablehnung gegen diese Hochsprachen, da man die Meinung vertrat, dass der durch Compiler übersetzte Programmcode zu hohe Performanceverluste einbrachte. Doch mit der Entstehung ausgereifter Entwicklungswerkzeuge und Compiler konnten die Gegner bekehrt werden und bald erkannte man, dass nur für wenige, spezielle Applikationen händisch programmierter Assemblercode notwendig war. Ein Großteil des Erfolges von Hochsprachen begründet sich auf folgenden Errungenschaften:

- Standards für C ermöglichen die Portabilität von Programmen zwischen verschieden Hardwareplattformen.
- Mit der Strukturierung des Programmcodes in Funktionen oder Subroutinen ist es möglich, die Systeme besser zu verstehen, zu entwickeln, zu warten und zu erweitern.
- Leistungsfähige Compiler erzeugen Assemblercode aus Hochsprachen, der mindestens äquivalente, doch meistens sogar bessere Performance erzielt als von Hand geschriebener Assemblercode.

Heutzutage werden in der Softwareentwicklung objektorientierte Sprachen wie Java, C++ oder Smalltalk eingesetzt, um die Datenstruktur und das Verhalten von Objekten in Klassen zu kapseln. Der nächste Abstraktionsschritt in Richtung Softwareplattformunabhängigkeit soll durch graphische Modellierungssprachen, wie z.B. *Unified Modeling Language* (*UML*), bzw. textbasierte Sprachen für die plattformunabhängige Spezifikation der Semantik von Aktionen, wie z.B. *Action Semantic Languages*, erreicht werden. Dieser Abstraktionsschritt ermöglicht die Erstellung von plattformunabhängigen Modellen, welche von der MDA gefordert wird. Zusammenfassend zeigt Abbildung 1.2 die geschichtliche Entwicklung der Abstraktionsebenen von Programmiersprachen.

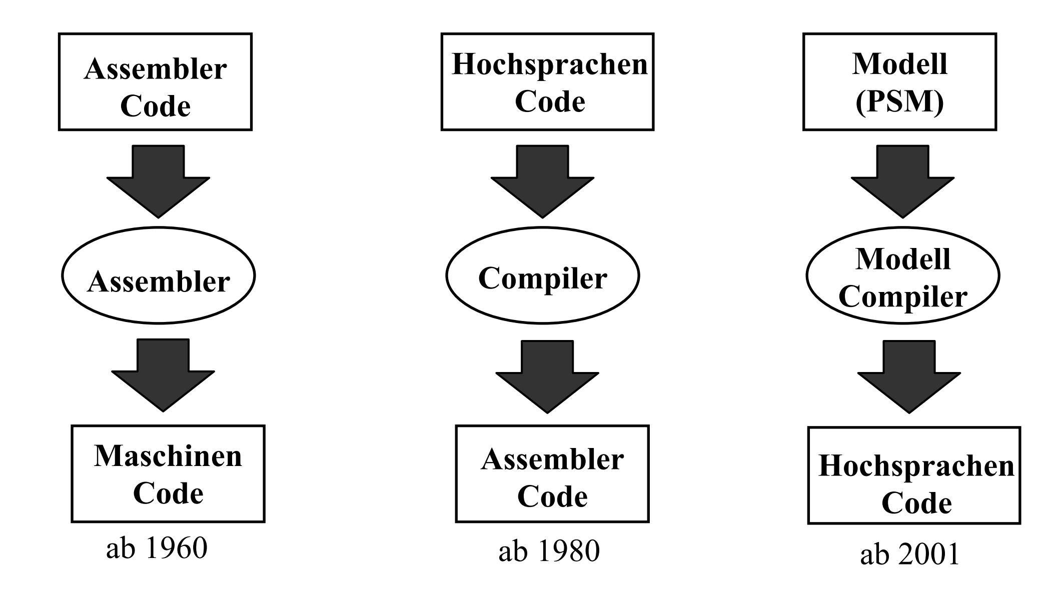


Abbildung 1.2 Programmiersprachen – Abstraktionsschritte basierend auf [Mell04]

Jeder Generationswechsel bringt mehrere Verbesserungen im Bereich Anforderungen an Programmiersprachen mit sich. Durch Abstraktion werden Lesbarkeit, Verständnis und Modifizierbarkeit von Applikationen verbessert. Mit dem Schritt zur graphischen Repräsentation von Software sollen Programme um einiges verständlicher werden.

Aus der Psychologie ist bekannt, dass Menschen nur bis zu sieben Dinge gleichzeitig wahrnehmen können. Deshalb ist es notwendig, mit wenigen Symbolen sehr viel an Information zu vermitteln. Um dies zu verdeutlichen, wird anschließend ein kleiner Ausschnitt eines Webshopsystems zuerst in Java (siehe nachfolgenden Code) als Beispiel für Hochsprachen und danach als UML Modell (Abbildung 1.3) repräsentiert. Durch dieses Beispiel sieht man sehr gut, dass die Repräsentationsform durch Modelle für den Menschen verständlicher wirkt als durch textbasierte Beschreibungen.

```
package webshop;

public class Kunde
{
  private int kundenNr;
  private java.lang.String vorname;
  private java.lang.String nachname;
  protected   java.util.Collection   dieBestellungen   =   new
     java.util.Vector();

  public Kunde(int kundenNr, java.lang.String vorname, ja
     va.lang.String nachname)
  {
    this.kundenNr = kundenNr;
    this.vorname = vorname;
    this.nachname = nachname;
```

```
  }
}

package webshop;

public class Bestellung
{
  private int bestellNr;
  private java.util.Date bestellDatum;
  protected webshop.Kunde derKunde;

  public Bestellung(int bestellNr, java.util.Date bestellDatum)
  {
    this.bestellNr = bestellNr;
    this.bestellDatum = bestellDatum;
  }
}
```



Abbildung 1.3 Webshop-Klassendiagramm

Code zu stark mit Infrastruktur verflochten

Die Spezifikation von Domänen, wie z.B. Bestellungssystem oder Lagerverwaltung, ist auf Codeebene stark mit der Infrastruktur der Softwareplattformen, wie z.B. Programmiersprachen, Datenbankzugriffe oder Betriebssystemdienste, vermischt. Dadurch ergibt sich folgendes Problem: Die Domänen bleiben über einen längeren Zeitraum konstant, hingegen sind Informationstechnologien eher kurzlebig und werden von performanteren Technologien abgelöst. Ein Umstieg auf neue Technologien ist mit hohen Kosten verbunden, da das Domänenwissen (wie z.B. Geschäftsprozesse) zu sehr mit den technischen Details der Softwareplattformen vermischt ist, wodurch das Domänenwissen schwer reproduzier- bzw. für die neuen Technologien nicht wiederverwendbar ist.

Für diese Problematik versucht die OMG mit MDA einen Lösungsweg anzubieten. Mit der Unterscheidung von plattformunabhängigen und plattformspezifischen Modellen ist es möglich, das Fachwissen über die Domäne von den technischen Details der Plattformen bzw. von den Programmiersprachen zu trennen. In plattformunab-

hängigen Modellen werden Geschäftsprozesse und Domänenwissen modelliert, ohne auf technische Details wie Programmiersprache, Softwareplattform, Hardware, Applikationsserver oder Betriebsystem einzugehen. Da keine technischen Details angegeben werden, beschreibt ein plattformunabhängiges Modell einen Geschäftsbereich bzw. einen allgemeinen Problembereich. [Sims04] beschreibt die Vision der OMG wie folgt:

The clues lie in these key parts of the vision statement: „separate business from technology", „enable intellectual property to move away from technology-specific code" and „transform fully specified PIMs on a range of platforms."

Durch automatische Transformationen ist es möglich, PIMs in PSMs überzuführen. Nach erfolgreicher Transformation beschreiben die plattformspezifischen Modelle die Geschäftsprozesse und das Domänenwissen in den Konstrukten der gewählten Softwareplattform. Aus einem plattformunabhängigen Modell können mehrere Implementierungen, die auf verschiedenartigen Plattformen wie J2EE, CORBA oder .NET beruhen, erstellt werden. Die Trennung von plattformunabhängigen und plattformspezifischen Modellen ermöglicht die Spezifikation von verschiedenen Problembereichen. Diese Spezifikationen können bei einem Softwaretechnologiewechsel wiederverwendet werden, um neue Applikationen – basierend auf der vorhandenen Problemstellung – durch automatische Codegenerierung erstellen zu können. Weiters befinden sich in den Modellen genügend Informationen, um Tests für das zu entwickelnde System automatisiert zu generieren.

Da Geschäftsbereiche über längere Zeit als konstant gelten (zumindest länger als die meisten Softwareplattformen), können die gespeicherten PIMs für neuere Implementierungen genützt werden. Im Laufe der Zeit verschwinden Plattformen vom Markt und wiederum entstehen neue Plattformen bzw. neue Versionen von Plattformen, wobei die Neuen beispielsweise durch bessere Performance und mehr Sicherheit ausgezeichnet sind und eine schnellere Entwicklung von Applikationen ermöglichen. Für eine Unternehmung ist die IT-Infrastruktur ein wichtiger Wettbewerbsfaktor und daher bleibt der laufende Umstieg auf neuere Technologien bzw. auf neuere Versionen der Technologien unausweichlich.

Geht eine Unternehmung nach dem MDA-Paradigma vor, stellt ein Technologiewechsel ein geringeres Problem dar, da nur eine Transformation zur Erstellung der neuen Applikation benötigt wird.

Die Entwicklung von neueren Versionen von Plattformen tritt bei weitem öfters auf als ein vollständiger Paradigmenwechsel. Abbildung 1.4 gibt einen Überblick über die zeitliche Entwicklung aktueller Softwareplattformen.

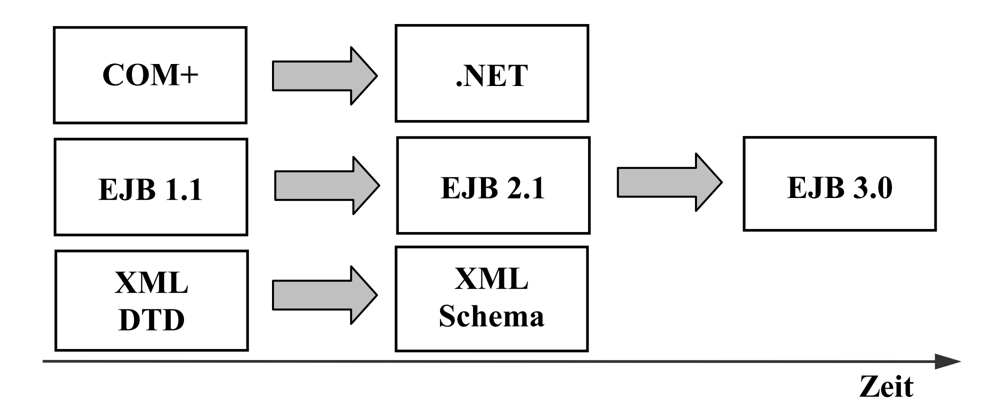


Abbildung 1.4.: zeitliche Entwicklung von aktuellen Plattformen

Aber auch die Evolution von Plattformen wird durch die korrekte Isolation der Domänenlogik von den technischen Spezifika gelöst. Somit bleiben mühsame Re-Implementierungen von vorhandenen Programmen erspart. Nur die Transformation eines plattformunabhängigen Modells in ein neues plattformspezifisches ist notwendig, um die performanteren Softwareplattformen nutzen zu können. Die Erstellung der Transformationen kann entweder firmenintern oder extern durch auf Transformationen spezialisierte Unternehmen erfolgen.

Wiederverwendung basiert auf einer zu feinen Form

Eine weitere potentielle Steigerung der Produktivität im Softwareentwicklungsbereich basiert auf der Wiederverwendung vorhandener Softwareartefakten. Dies wurde in der Informatik schon früh erkannt und im Laufe der Zeit wurde eine Reihe von Strukturierungsmechanismen entwickelt. Im Folgenden werden diese Mechanismen beschrieben.

Als erste Reaktion wurden *Funktionen* eingeführt. Somit konnten Programme in Funktionen strukturiert werden und diese Funktionen konnten entweder im selben Programm öfters verwendet werden oder in anderen Programmen – durch die Erstellung von Bibliotheken – wiederverwendet werden.
Die Strukturierung von Programmen in Funktionen hat den großen Vorteil, dass Änderungen in Funktionen nur einmal und nicht an jeder Stelle, wo diese verwendet werden, vorgenommen werden müssen. Durch Funktionen wird der Code um einiges übersichtlicher und dadurch auch besser wartbar. Doch leider entsteht durch die

Verwendung von Funktionen ein neues Problem. Die globalen Datenstrukturen sind mit den Funktionen nicht gekoppelt, sondern sind global von jeder Funktion erreichbar. Da nun mehrere Funktionen unkontrollierten Zugriff auf globale Datenstrukturen haben können, müssen bei einer Änderung in der globalen Datenstruktur mehrere Funktionen modifiziert werden, auch wenn diese Änderung nur wegen einer einzigen Funktion notwendig ist.

Die Antwort auf letztgenanntes Problem waren *Objekte*. Diese ermöglichten die Kapselung von Daten und ihren zugehörigen Funktionen. Somit stieg auch das Wiederverwendungspotential stark an. Doch schon bald erkannte man, dass auch Objekte nicht die Fähigkeit besaßen, die erhöhte Komplexität von verteilten Anwendungen zu kompensieren. Gerade im Bereich Geschäftsapplikationen, der einen Großteil von Softwareapplikationen repräsentiert, wurde das Bedürfnis nach kostengünstigerer und effizienterer Programmierung (kürzere Time-to-Market) sichtbar. Diese Erkenntnis war die Geburtsstunde der Komponenten.

Eine *Komponente* ist eine Menge von eng zusammenarbeitenden Objekten, die mit einer Menge von wohldefinierten Schnittstellen zur Außenwelt gekoppelt wird. Mit dem Einzug von Komponenten wurden *Enterprise Java Beans* (*EJBs,* [SUN03b]) äußerst erfolgreich und mit ihnen wurden Applikationsserver populär. Um die Funktionsweise von Komponenten und Applikationsserver (wie z.B. Jboss, BEA WebLogic) zu verdeutlichen, soll folgendes Beispiel dienen:

Jeder CD-Player liest und spielt CDs, da ein einheitlicher CD-Standard existiert. Im Informatikbereich könnte ein Applikationsserver mit einem CD-Player verglichen werden und eine Komponente mit einer CD, denn jede Komponente läuft in jeden beliebigen Applikationsserver.

Die Einführung von Komponenten löste viele Probleme der Softwareentwicklung, leider brachte sie aber ein neues Problem mit sich. Auch Komponenten unterliegen Änderungen im Laufe der Zeit. Wenn sich nun ein Interface ändert, müssen dadurch weitere Modifikationen im Quelltext innerhalb von Komponenten stattfinden. Danach bedarf es an Komponententests und schließlich an Systemtests. Dies könnte einen Grund dafür darstellen, warum Komponenten bei weitem nicht so stark eingesetzt wurden als erwartet.

Durch den Einsatz von *Domänenmodellen* wird das Wiederverwendungspotential weiter gesteigert. Domänenmodelle sind Modelle, die eine bestimmte Anwendungs-

domäne plattformunabhängig beschreiben. Viele Anwendungsdomänen von Software, wie z.B. Bestell- oder Lagerverwaltungssysteme, bleiben über längere Zeit konstant. Werden für diese Domänen plattformunabhängige Modelle entwickelt, so kann die Entwicklung von Applikationen stark beschleunigt werden. Eine durch plattformunabhängige Modelle definierte Domäne kann in mehreren Applikationen eingesetzt werden. Ein Domänenmodell sollte die Struktur und das Verhalten einer Domäne definieren. Für die Strukturmodellierung bieten sich Klassendiagramme an, für die Verhaltensmodellierung Aktivitätsdiagramme. Die erstellten Domänenmodelle werden in Repositories gespeichert und können als Schablonen in den Modellierungswerkzeugen verwendet werden. In [Arlo04] werden bereits einige Domänen durch sogenannte *Archetype Patterns* spezifiziert und die OMG arbeitet intensiv an der Erstellung von Domänenspezifikationen[1]. Das Ziel der OMG ist die automatisierte Erstellung von PSMs aus den definierten Domänenmodellen.

Abbildung 1.5 gibt zusammenfassend einen Überblick über die besprochenen Konzepte und ihre Wiederverwendungspotentiale.

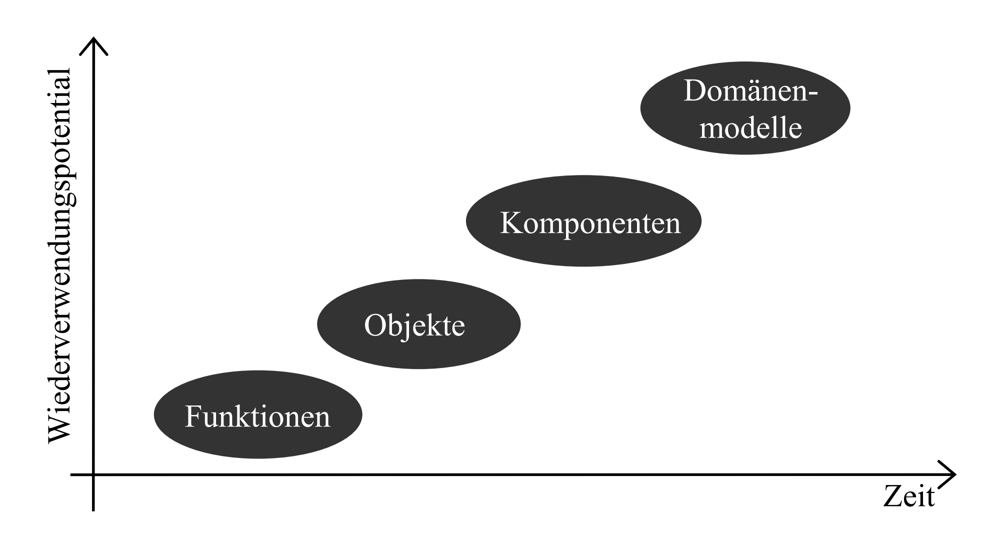


Abbildung 1.5.: Wiederverwendungspotential von Software

„Business to Code Mappings“ werden nicht archiviert

Bei der Erstellung einer Applikation wird überlegt, wie der Problemraum in Programmcode (Lösungsraum) abgebildet werden kann. Diese Überlegungen werden aber nicht separat gespeichert, sondern gehen nach dem Erstellen des Codes wieder verloren bzw. werden mit den plattformspezifischen Details im Code verflochten.

[1] http://www.omg.org/technology/documents/domain_spec_catalog.htm (Stand: 12.2.2005)

Danach ist der Problemraum zwar implizit im Programmcode enthalten, aber um diesen rekonstruieren zu können, bedarf es eines mühsamen Re-Engineerings.

Auch hierfür bietet die MDA einen Lösungsweg an. Die Applikation in einem PIM zu modellieren, sollte die erste Aufgabe in einem MDA-Projekt darstellen. Das PIM beschreibt alle funktionalen Aspekte der Applikation, aber ohne auf plattformspezifische Details einzugehen. Diese werden erst durch eine Transformation des PIMs in ein PSM eingebracht. Diese Transformationen sind großteils äquivalent zu den Überlegungen des Programmierers beim Erstellen der Applikation.

1.2 Motivation für diese Arbeit

Wie Berichte der Österreichischen Eisenbahnen und der Deutsche Bank Bauspar AG belegen, wurden in der Praxis MDA-Projekte bereits erfolgreich durchgeführt. Details zu diesen Projekten findet man auf der OMG Website unter MDA Success Stories[1]. In diesen Berichten vermitteln die OMG und MDA-Werkzeughersteller das Bild, dass durch den Einsatz von MDA-Werkzeugen einsatzfähige Anwendungen auf Knopfdruck automatisch erzeugt werden können. Weiters wird versprochen, dass bei der Erstellung von Anwendungen keine Kenntnisse über Plattformen, wie z.B. Programmiersprachen, Betriebsysteme, Applikationsserver oder Datenbanken, benötigt werden und dadurch Software auch von Nicht-InformatikerInnen entwickelt werden kann. In den Berichten werden aber keine Angaben über relevante Probleme von Modellierungssprachen (wie Mängel in der Verhaltensmodellierung von UML) oder Transformationssprachen (noch keine standardisierte Sprache vorhanden) angeführt.

Diese Versprechungen erzeugen hohe Erwartungen bei SoftwareentwicklerInnen und ProjektleiterInnen, da sich diese durch den Einsatz von MDA-Werkzeugen enorme Produktivitätssteigerungen erhoffen. Ohne Vorkenntnisse über MDA ist die Auswahl des einzusetzenden Werkzeuges äußerst schwer zu treffen, da eine Vielzahl an Werkzeugen angeboten wird und sich diese stark im Funktionsumfang unterscheiden. Einige Werkzeuge erzeugen einen Großteil der Applikationen für eine bestimmte Plattform oder für einen bestimmten Einsatzbereich, doch im Allgemeinen werden nur Codefragmente erstellt, die händisch ausprogrammiert werden müssen. Durch diese Vielfalt von MDA-Werkzeugen besteht die Gefahr, dass unpassende Werkzeuge für Projekte eingesetzt werden und dadurch das Risiko für ein Scheitern des Projektes stark ansteigt.

---

[1] http://www.omg.org/mda/products_success.htm (Stand: 1.1.2005)

Um einerseits eine exaktere Vorstellung von dem Begriff MDA zu bekommen und andererseits den aktuellen Stand der Technik zu bestimmen, wird eine konkrete Fallstudie durchgeführt, in der ein MDA-Werkzeug für die Erstellung einer mehrschichtigen Webapplikation eingesetzt wird. Weiters werden durch die Fallstudie Bereiche der Softwareentwicklung aufgezeigt, die durch den Einsatz von aktuellen MDA-Werkzeugen stark profitieren, außerdem wird auch geklärt, welche Bereiche noch Forschungsaktivitäten benötigen. Durch die Kategorisierung von MDA-Werkzeugen und durch die in einem Kriterienkatalog gesammelten Anforderungen an MDA-Werkzeuge wird eine erste Entscheidungshilfe geboten, um zu bestimmen, welche Werkzeuge für den Einsatz in einem bestimmten Projekt in Frage kommen.

Weiters erleichtert ein verbessertes Verständnis über MDA die Auswahl eines geeigneten MDA-Werkzeuges. Werkzeuge, die die speziellen Anforderungen eines Projektes unterstützen, optimieren bzw. automatisieren häufig auftretende Arbeitsschritte und entlasten dadurch ProgrammierInnen von Routinetätigkeiten. Durch die Sensibilisierung der Anwender, hinsichtlich der signifikanten Unterschiede im Funktionsumfang der MDA-Werkzeuge, wird auch die langfristige Entwicklung von MDA gesichert, da keine unerfüllbaren Erwartungen bei den AnwendernInnen geweckt werden.

1.3 Zielsetzung

Diese Diplomarbeit soll die theoretischen Grundlagen der MDA und ihre praktische Umsetzung durch konkrete Technologien erklären. Der Leser soll durch diese Arbeit Antworten auf folgende Fragen bekommen:

- *Was sind Modelle und wie können diese effizient zur Codegenerierung genutzt werden?*
- *Welche verschiedenen Modellarten können im MDA-Paradigma unterschieden werden und wie werden diese in weitere Modellarten transformiert?*
- *Welche Technologien bzw. Standards sollen in einem MDA-Projekt eingesetzt werden?*
- *Welche Anforderungen sollten MDA-Werkzeuge erfüllen?*
- *Welche MDA-Werkzeuge haben sich am Softwaremarkt etabliert?*
- *Wie unterscheidet sich ein MDA-Softwareprojekt von einem „gewöhnlichen“ Softwareprojekt?*

Die Hauptforschungsfragen dieser Diplomarbeit sind einerseits die Erstellung eines Kriterienkataloges für die Evaluierung von MDA-Werkzeugen und andererseits die Eruierung des aktuellen technischen Stands der Werkzeuge. Dabei soll untersucht werden, ob und wie sich MDA-Werkzeuge von gewöhnlichen Codegeneratoren unterscheiden.

1.4 Aufbau

In dieser Diplomarbeit wird ein fundierter Überblick über die Grundlagen und die theoretischen Konzepte von Model Driven Architecture gegeben. Diese Aufgabe nimmt die erste Hälfte der Arbeit ein. Der zweite Teil der Arbeit beschäftigt sich mit dem praktischen Einsatz des MDA-Paradigmas in der Softwareentwicklung. Dabei werden MDA-Werkzeuge anhand eines Kriterienkataloges besprochen. Abschließend wird der heutige technische Stand der Werkzeuge anhand des Fallbeispiels CALENDARIUM bestimmt. Im Folgenden wird ein Überblick über die Kapitel dieser Arbeit und deren Inhalt gegeben.

Das zweite Kapitel beschäftigt sich mit den Grundlagen von Model Driven Architecture. Dem Leser wird ein Überblick über Konzepte, wie Modelle, Metamodelle, Transformationen sowie den grundsätzlichen Aufbau des MDA-Paradigmas geboten. Zusätzlich werden die Vorteile des Einsatzes von Softwaremodellen besprochen. Da MDA keine Technologie im „engeren" Sinne darstellt, werden die einsetzbaren Metamodelle, wie UML oder CWM, und weitere Technologien, wie z.B. XMI oder UML Profiles, grob skizziert.

Im Kapitel 3 werden am Beginn die Aktivitäten eines MDA-Entwicklungsprozesses identifiziert und anhand dieser Aktivitäten Kriterien für die Evaluierung von MDA-Werkzeuge abgeleitet. Abschließend werden die erhältlichen MDA-Werkzeuge kategorisiert und eine kleine Marktübersicht geboten.

Kapitel 4 beschreibt die bekanntesten Vertreter von kommerziellen und Open Source Werkzeugen. Abschließend wird ein Werkzeugvergleich anhand des in Kapitel 3 definierten Kriterienkataloges angeführt. Damit sollen Stärken und Schwächen der Werkzeuge einfach zu eruieren sein.

Im Kapitel 5 wird anhand einer Fallstudie gezeigt, inwieweit die heutigen Technologien den von der OMG propagierten MDA-Ansatz umsetzen. Als Beispielimplementierung dient ein Auszug aus dem CALENDARIUM-Projekt. Die Applikation wird mittels Enterprise Java Beans, relationaler Datenbank, JSPs und Servlets realisiert. Als MDA-Werkzeug wird ArcStyler eingesetzt, um aus einem PIM ein lauffähiges System zu generieren.

Im letzten Kapitel dieser Diplomarbeit werden eine Zusammenfassung der Werkzeugevaluierung und ein Überblick über die Erfahrungen der Fallstudie gegeben. Weiters wird ein kurzer Ausblick auf die Entwicklungsmöglichkeiten von MDA gegeben.

Da die für die Fallstudie erstellten Modelle und der aus den Modellen abgeleitete Code äußerst umfangreiche Dokumente darstellen, ist dieser Diplomarbeit eine CDROM beigelegt, die das gesamte Modell und den gesamten Code der Terminverwaltung beinhaltet. Im Anhang A werden die Pfade zu den Modellen angegeben und im Anhang B die Pfade zu den Quelltexten und zu den lauffähigen Komponenten.

Kapitel 2

MDA - Model Driven Architecture

Im Gegensatz zum Kapitel 1.1, in dem die modellgetriebene Entwicklung allgemein betrachtet wird, beschreibt dieses Kapitel die theoretischen Konzepte und Grundlagen von MDA. MDA, die OMG-Version der modellgetriebenen Softwareentwicklung, basiert auf den OMG-Standards. Deshalb wird im Unterkapitel 2.1 die geschichtliche Entwicklung der Ziele der OMG beschrieben und ein Überblick über die wichtigsten OMG-Standards gegeben. Modelle stehen im Mittelpunkt des MDA-Paradigmas, deshalb werden im Unterkapitel 2.2 die Begriffe Modell und Metamodell erklärt. Abschließend werden im Abschnitt 2.3 die unterschiedlichen Modellebenen von MDA und Transformationen besprochen.

2.1 Überblick

Die Object Management Group verfolgte über mehrere Jahre die Standardisierung des *Object Request Broker* (*ORB*). Diese Aktivität war ein Bestandteil der *Object Management Architecture* (*OMA*) [OMG95], die ein Framework für verteilte Systeme, die auf der *Common ORB Architecture* (*CORBA*) basieren, darstellt. Ab 1995 begann die Adaption von industriespezifischen Spezifikationen. Weiters wurde mit dem Einzug der objektorientierten Programmiersprachen die Forderung nach einer standardisierten Modellierungssprache für objektorientierte Systeme stärker. Aus dieser Bewegung heraus entwickelte sich die *Unified Modeling Language* (*UML*), die heute in Version 1.5 [OMG03b] vorliegt. Im Jahr 2001 wurde ein weiteres Framework von der OMG veröffentlicht, nämlich die *Model Driven Architecture* (*MDA*) [OMG01], [OMG03a]. Im Gegensatz zu OMA oder CORBA, welche nur auf

verteilte Systeme ausgerichtet sind, fokussiert MDA auf die modellgetriebene Entwicklung von Softwaresystemen.

Im Folgenden wird anhand des Logos von MDA (Abbildung 2.1) ein Überblick über die Grundlagen von MDA gegeben.

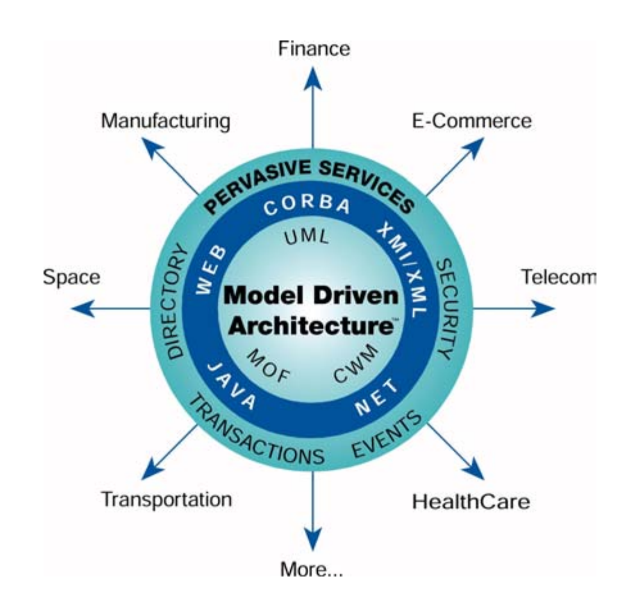


Abbildung 2.1: MDA-Logo

Das Logo kann grob in vier Bereiche eingeteilt werden. Der innerste Kreis des Logos listet die wichtigsten Technologien für die Modellierung von Softwaresystemen auf. Diese sind Unified Modeling Language (UML), *Meta Object Facility* (*MOF*) [OMG02c], *Common Warehouse Model* (*CWM*) [OMG03c]. Diese Technologien stellen bereits existierende OMG-Standards dar. UML wird für die Modellierung von Struktur und Verhalten von Systemen eingesetzt und bietet dafür unterschiedliche Diagrammarten wie Klassen-, Aktivitäts-, Sequenzdiagramme und noch viele mehr. Für die Modellierung von Metasprachen (z.B. UML) wird MOF eingesetzt. Weiters sind Schnittstellen für die Manipulation von MOF-basierten Modellen und ihren Instanzen definiert. Diese Schnittstellen werden entweder durch CORBA IDL oder Java beschrieben. Für den Bereich Data Warehouse wurde von der OMG ein eigenes Metamodell namens CWM entwickelt. CWM unterstützt den gesamten Lebenszyklus vom Entwurf über die Entwicklung bis zur Wartung der Data Warehouse-Applikationen. Ein weiterer Einsatzbereich von CWM stellt die Modellierung von Datenbankschemata für (Data Warehouse-) Applikationen dar.
In der Abbildung 2.1 werden zwar einige Technologien der OMG angegeben, jedoch gibt es noch weitere die enormes Potential für MDA besitzen. Zu diesen Technologien zählen *Object Constraint Language* (*OCL*) [OMG03d] und *UML Profiles*.

OCL erlaubt die textuelle Spezifikation von Einschränkungen, die durch grafische Notationen nicht oder nur schwer dargestellt werden können. UML ist durch UML Profiles entweder für bestimmte Softwarebereiche, wie z.B. verteilte Systeme oder Echtzeitsysteme, oder für bestimmte Softwareplattformen, wie z.B. EJB oder CORBA, erweiterbar. Erweiterungen durch UML Profiles verändern aber nicht das UML-Metamodell. Somit ist die Standardisierung und Interoperabilität von UML sichergestellt. Folgende UML Profiles wurden bereits von der OMG standardisiert:

- *UML Profile for Corba* [OMG00]: Dieses Profile ermöglicht die Transformation eines PIMs in ein CORBA-spezifisches PSM.
- *UML Profile for EDOC* [OMG04a]: Enterprise Distributed Object Computing Profile wird für die plattformunabhängige Modellierung von komponentenbasierten und verteilten Systemen eingesetzt.
- *UML Profile for EAI* [OMG04b]: Enterprise Application Integration Profile unterstützt die Erstellung von *lose verbundenen Systemen*, die über asynchrone oder nachrichtenbasierte Methoden kommunizieren.
- *UML Profile for Schedulabitlity, performance, and time* [OMG05]: Dieses Profile wird für die Modellierung von zeitlichen und leistungsbezogenen Aspekten der Echtzeitanwendungen eingesetzt.
- *UML Profile for QoS and Fault Tolerance* [OMG04c]: Durch dieses Profile wird UML durch Quality of Service und fehlertoleranten Konzepten erweitert.
- *UML Profile for Java and EJB* [OMG04d]: Dieses Profile befindet sich im EDOC UML Profile und ermöglicht die Erstellung von Java-basierten PSMs.
- *UML Testing Profile* [OMG03e]: Dieses Profil definiert eine Modellierungssprache für den Entwurf, die Spezifikation und die Dokumentation von Testsuits.

Die Trennung von der Problemstellung und der plattformspezifischen Implementierung wird auch im Logo von MDA angedeutet. Rund um die Modellierungstechnologien werden aktuell genutzte Technologien aufgelistet, die Zielplattformen für die Modelltransformationen darstellen. Die wichtigsten Softwareplattformen sind Common Object Request Broker Architecture (CORBA), J2EE und .NET. Für den Modellexport bzw. für die Modellspeicherung wird *XML Metadata Interchange* (*XMI*) [OMG02b] als Plattform verwendet. Durch XMI können XML Schemata und DTDs von MOF-basierenden Metamodellen abgeleitet werden und weiters Instanzen von MOF-basierenden Metamodellen als XML-Dokumente exportiert bzw. gespeichert werden. Die aufgelisteten Plattformen spezifizieren keine abgeschlossene Menge

von Zieltechnologien der MDA, sondern zeigen nur häufig genutzte Beispieltechnologien. Natürlich liegen ältere Technologien, wie z.B. COBOL oder FORTRAN, genauso im Fokus der modellgetriebenen Entwicklung, wie neu entstehende Technologien.

Der äußere Ring beinhaltet sogenannte *unterstützende Services* (*Pervasive Services*). Diese Services sind grundlegende Dienste, die in einem PIM eingearbeitet werden und danach durch Transformationen in beliebige Implementierungen realisiert werden. Für verteilte Systeme sind vor allem unterstützende Services in den Bereichen Sicherheit, Transaktionen und Persistenz relevant. Die OMG arbeitet intern an der Erstellung von PIMs für CORBAservices[1], um diese Dienste plattformneutral zur Verfügung zu stellen.

Die Vektoren am Rand der Abbildung 2.1 geben einen Überblick über die verschiedenen Märkte und Domänen, die vermutlich einen großen Nutzen aus den modellgetriebenen Entwicklungen ziehen werden.

2.2 Modelle und Metamodelle

Modelle sind das Schlagwort von MDA. Um die unterschiedlichen Arten der Modelle des MDA-Paradigmas zu erläutern, muss zuvor geklärt werden, was eigentlich der Begriff Modell in der Informatik bedeutet. Außerdem muss festgelegt werden, wie ein Modell durch ein Metamodell beschrieben und was unter dem Begriff Metamodell verstanden wird.

Modelle

Ein Modell ist eine Abstraktion von einer in der Realität existierenden Entität. Modelle werden verwendet, um uninteressante Details zu verschweigen und wichtige Aspekte ins Rampenlicht zu rücken. Folgende Eigenschaften (basierend auf [Seli03]) charakterisieren Modelle nicht nur im Softwarebereich:

- *abstrakt*: Modelle abstrahieren von irrelevanten Details und stellen relevante Aspekte in den Mittelpunkt.

[1] http://www.omg.org/technology/documents/formal/corbaservices.htm (Stand 14.2.2005)

- *einfach*: Modelle besitzen durch grafische Notationselemente eine menschenfreundliche Repräsentationsform.

- *präzise*: Durch formale Modellierungssprachen können präzise und berechenbare Modelle erstellt werden.

- *voraussagend*: Durch Modelle können Probleme schon im Vorfeld erkannt werden, wodurch auf diese besser reagiert werden kann.

- *günstig*: Modelle sind um einiges kostengünstiger in der Erstellung als die konkrete Realisierung.

Aus den aufgelisteten Eigenschaften ergeben sich mehrere Vorteile, die für einen Einsatz von Modellen bei der Erstellung von Softwaresystemen sprechen. Modelle helfen, die Struktur und das Verhalten eines Systems zu spezifizieren und zu kommunizieren. Existiert bereits ein Softwaresystem und gilt es, dieses zu erweitern, so trägt eine Dokumentation in Form von Modellen zum besseren Verständnis des Systems und zur rascheren Erweiterung bei. In jedem Softwareentwicklungsprozess wird versucht, Fehler frühzeitig zu entdecken. Bei dem Einsatz von Modellen ist eine frühe Simulation von Prototypen möglich. Fehler können auf einer abstrakteren Ebene schneller und günstiger ausgemerzt werden als auf Quelltextebene. Außerdem helfen Modelle bei der Implementierung des Systems durch die automatische Generierung von Skeleton-Code und Templates. Dieser Ansatz der Entwicklung von Softwaresystemen wird durch die MDA fortgesetzt, um die automatische Erstellung von lauffähigen Applikationen aus Modellen zu ermöglichen.

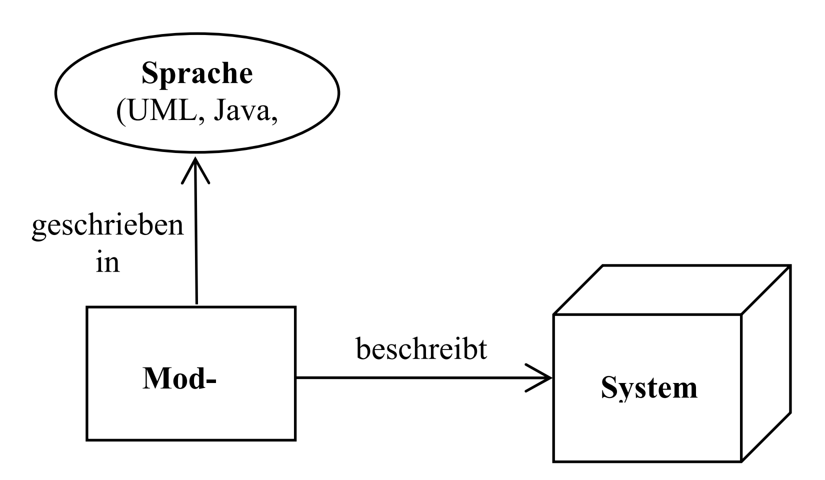


Abbildung 2.2: Definition und Aufgabe von Modellen

Doch heutzutage bestehen noch einige Limitationen bei dem Einsatz von Modellen in der Softwareentwicklung und somit steht die Softwareentwicklung noch weit hinter reiferen Industrien, wie der Automobilindustrie oder der Elektrotechnik. Sogar im

Hardwaredesign werden Modelle bei weitem effizienter eingesetzt als im Softwaredesign. Folgende Probleme behindern einen effizienteren Einsatz von Modellen in der Softwareentwicklung:

- Modelle werden nur als Dokumentation verwendet
- Unterschiede zwischen Modell und Code
    - Änderungen im Modell werden auf Codeebene nicht automatisch übertragen und vice versa. Der letzte Beisatz verursacht vom Code abweichende Dokumentationen, da am Ende eines Projektes wegen Zeitdruck viele Änderungen nur mehr auf Codeebene durchgeführt werden.
- Keine Modelltransformationen
    - Keine ausgereiften, standardisierten Transformationssprachen
    - Nur geringe Werkzeugunterstützung

Um die Verwendung von Modellen im Softwaredesign voranzutreiben und die vorher erwähnten Limitationen zu beheben, bedarf es an wohldefinierter Modelle. Modelle im Sinne der MDA sind alle im Zuge eines Entwicklungsprozesses entstandenen wohldefinierten Modelle des zu entwickelnden Softwaresystems. Diese Modelle können auch unterschiedliche Sichtweisen auf ein Softwaresystem ermöglichen. Da Modelle in verschiedenen Varianten auftreten, fällt eine exakte Definition schwer. Darum werden vorerst einige in der MDA-Literatur verwendete Definitionen angeführt:

- Eine Beschreibung eines Systems (oder Teile davon), geschrieben in einer wohldefinierten Sprache. Eine wohldefinierte Sprache ist eine Sprache mit wohldefinierter Form (Syntax) und Bedeutung (Semantik), die für die automatische Verarbeitung durch Computer geeignet ist. [Klep03]

- Eine Repräsentation eines Teils der Funktion, der Struktur oder des Verhaltens eines Systems. [OMG01]

- Eine Beschreibung oder Spezifikation eines Systems und seiner Umgebung für einen bestimmten Zweck, wie z.B. die Spezifikation des Verhaltens oder der Datenstruktur. Ein Modell wird oft durch die Kombination von grafischen Notationselementen und Texten repräsentiert. [OMG03a]

Im Folgenden werden Gemeinsamkeiten der in der Literatur verwendeten Definitionen zusammengefasst, um eine umfassende Definition für den Begriff Modell im MDA-Umfeld zu geben.
Modelle werden immer durch Sprachen definiert. Eine Sprache kann UML, eine Programmiersprache oder auch jede natürliche Sprache sein. Um automatische Transformationen zu ermöglichen, muss in der modellgetriebenen Softwareentwicklung die Definition für den Begriff Modell eingeschränkt werden. Ein Modell muss in einer wohldefinierten Sprache angegeben werden, da diese vom Computer verarbeitet werden muss, um vollständig automatisierbare Modelltransformationen realisieren zu können. Natürliche Sprachen, wie z.B. Deutsch oder Englisch, eignen sich nicht für die Spezifikation eines Systems, da diese vom Computer nicht exakt verarbeitet werden können. Diese Problematik wird bereits bei der Benutzung von Übersetzungsprogrammen sichtbar.

Außer der Forderung nach Wohldefiniertheit gibt es keine weitere Restriktion der Syntax von Modellierungssprachen. Somit ist Quelltext, wie z.B. Java oder C++, ebenso wie ein graphisches Modell, wie z.B. ein UML-Klassendiagramm oder ein ER-Diagramm, ein Softwaremodell. Natürlich stellt ein Quelltext ein äußerst plattformspezifisches Modell dar.
Wie nun wohldefinierte Modelle spezifiziert werden, zeigt der nächste Abschnitt über Metamodelle.

Metamodelle

Im Abschnitt Modelle wurde ein Modell als Beschreibung eines Systems in einer wohldefinierten Sprache, die vom Computer exakt verarbeitet werden kann, definiert. Dabei wurde die wohldefinierte Sprache als gegeben angenommen. In diesem Abschnitt ist die Beschreibung einer solchen wohldefinierten Sprache von Interesse, um die weiteren Zusammenhänge des MDA-Ansatzes verstehen zu können.

Programmiersprachen werden im Allgemeinen durch Grammatiken definiert, welche mögliche Varianten von Zeichenfolgen mittels Token bestimmen. Durch einen Parser wird überprüft, ob die Syntax der Anweisungen gültig ist. Als Definition von Sprachen sind Grammatiken nur für textbasierte Sprachen relevant. Da Modellierungssprachen im Allgemeinen nicht auf Text basieren, sondern über grafische Elemente der Systembeschreibung verfügen, wird ein eigener Mechanismus benötigt, um Modellierungssprachen im MDA-Bereich zu definieren. Da UML die Hauptmo-

dellierungssprache der modellgetriebenen Softwareentwicklung darstellt, wird im Folgenden der Metamodellierungsansatz der OMG vorgestellt.

Ein Modell definiert welche Elemente in einem System existieren dürfen. Angenommen wir modellieren die Klasse *Termin* innerhalb eines Kalendersystems, dann darf eine Instanz *Arzttermin* vom Typ *Termin* zur Laufzeit des Systems auftreten. Nun gehen wir eine Ebene höher. Warum ist es überhaupt erlaubt, im Modell eine Klasse zu modellieren? Da UML durch ein Metamodell beschrieben wird und in diesem ein Konstrukt Klasse existiert, darf eine Klasse *Termin* in unserem eigenen Modell modelliert werden. Diese Klasse *Termin* stellt eine Instanz der Klasse *Class* des UML-Metamodells dar. Das UML-Metamodell definiert somit die Syntax und Semantik von Klassen durch die Klasse *Class*.

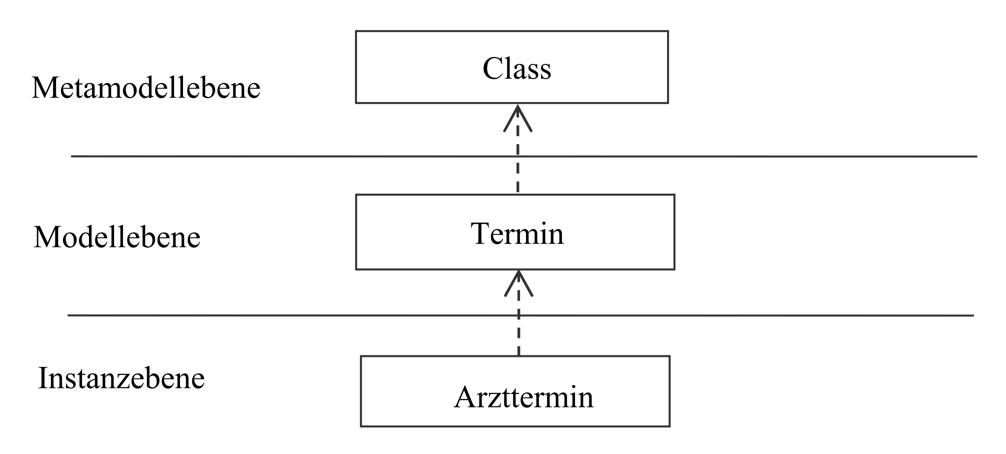


Abbildung 2.3: Metamodellierung

Da ein Metamodell wie UML wieder durch ein Modell beschrieben wird, muss dieses Modell selbst wieder in einer wohldefinierten Sprache, bezeichnet als Metasprache, deklariert werden. Dabei müssen alle Elemente die zur Spezifikation von UML oder anderen Metamodellen notwendig sind in dieser Metasprache vorhanden sein. Von der OMG wird Meta Object Facility (MOF) als Metasprache für UML und für alle anderen Metamodelle der OMG, wie z.B. Common Metadata Warehouse (CWM), verwendet.

Es besteht ein großer Unterschied zwischen einer Sprache wie UML und einer Metasprache wie MOF. Letztgenannte dient primär der Beschreibung von Modellierungssprachen und nicht wie UML zur Spezifikation von Softwaresystemen. Metasprachen werden aber genauso wie Sprachen durch ein Modell beschrieben. Deshalb wird wieder eine wohldefinierte Sprache benötigt, welche das Modell der Metasprache definiert. Somit würden sich aber unendlich viele Metamodellierungsschichten auftun, da immer wieder eine Sprache zur Beschreibung der obersten Schicht benö-

tigt würde. Dieses Problem wird im Metamodellierungsansatz der OMG dadurch vermieden, dass die Metasprache reflexiv (d.h. sich selbst erklärend) ist. Um diesen Lösungsweg zu verdeutlichen, soll die Deutsche Sprache als Beispiel dienen. Deutsch wird durch Grammatikregeln und einem Wortschatz (Wörterbuch) definiert, wobei die Grammatikregeln und die Wörter selbst mit Deutsch beschrieben werden. Weiters werden Sprachen und Metamodelle bzw. Metasprachen und Metametamodelle als selbiges angesehen, da die Unterscheidung keinen praktischen Nutzen bringt.

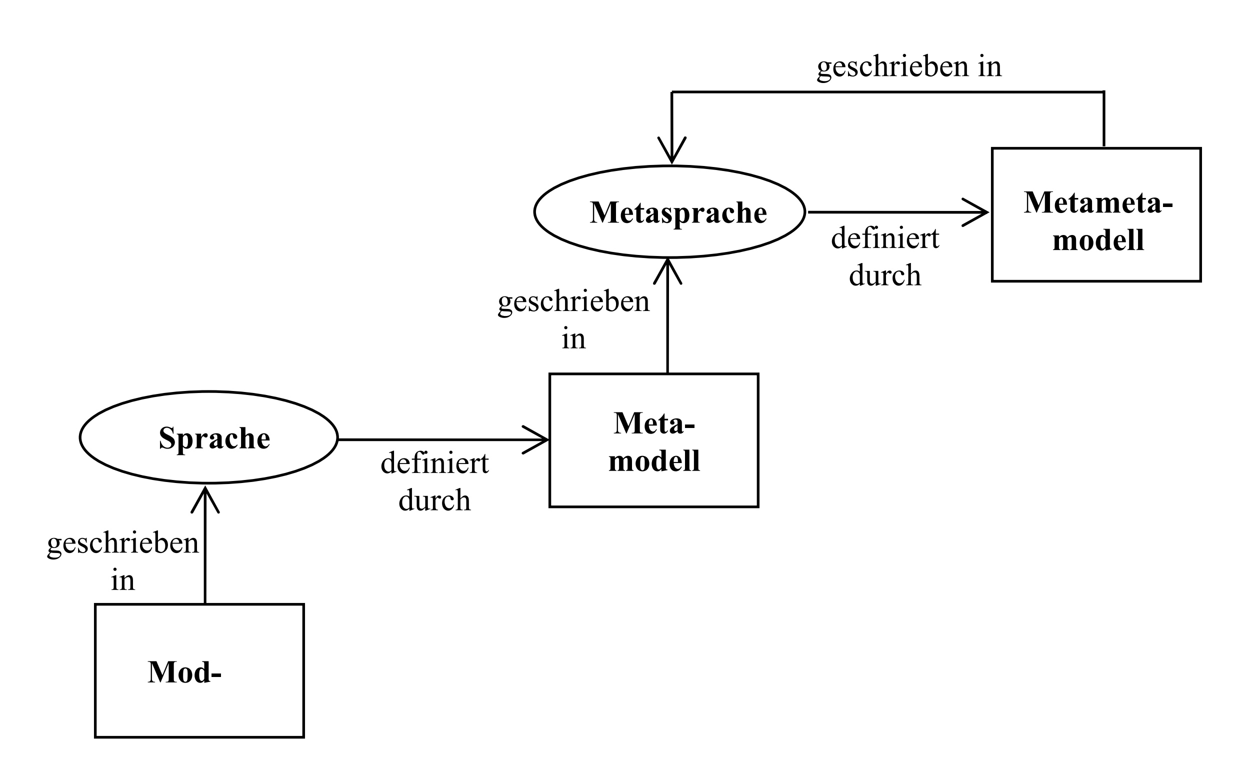


Abbildung 2.4: Sprachen und Metasprachen

Da wir nun das nötige Grundwissen über Metamodellierung besitzen, können wir den Metamodellierungsansatz der OMG genauer begutachten. Von der OMG wird MOF als Metamodellierungssprache zur Definition von Metamodellen verwendet. Konkrete Instanzen von MOF werden als Modellierungssprachen eingesetzt. Abbildung 2.5 gibt einen Überblick über den Aufbau des OMG Ansatzes. Im Folgenden werden die vier Ebenen der Metamodellierung anhand von MOF und UML genauer beschrieben.

M0-Ebene: Instanzen

Elemente der M0-Ebene repräsentieren reale Entitäten oder ihre entsprechenden Abbildungen in der Software. Als Beispiel könnte ein Arzttermin am 1.1.2004 um 13

Uhr fungieren. Die Eigenschaften von Terminen werden eine Ebene höher (d.h. auf der M1-Ebene) modelliert.

M1-Ebene: Modell des zu beschreibenden Systems

Auf dieser Ebene befinden sich Modelle von Softwaresystemen, welche z.B. in UML oder CWM modelliert werden. Die Konzepte auf der M1-Ebene stellen Kategorisierungen von Instanzen der M0-Ebene dar. Ein Element der M0-Ebene wird durch ein Element der M1-Ebene repräsentiert. Diese Beziehung wird in der Abbildung 2.5 mit dem Stereotyp *«representedBy»* notiert.

M2-Ebene: Metamodell

Die Elemente der M1 Ebene sind Instanzen von Elementen der M2 Ebene. Diese Beziehungen wird in der Abbildung 2.5 mit dem Stereotyp *«instanceOf»* notiert. M2-Elemente kategorisieren M1-Elemente. Die Unterteilung in Ebenen hilft die Aufgaben der Elemente zu trennen. Auf M1 müssen nur die Konzepte definiert werden, welche für Instanzen der M0-Ebene relevant sind. Ebene M2 beinhaltet wiederum nur Konzepte (z.B. *Class* und *Association*), die auf M1 verwendet werden. Die M2-Ebene wird auch Metamodellebene genannt, da auf dieser Ebene die Modellierungssprachen der OMG definiert werden.

M3-Ebene: Metametamodell

Die Elemente der M2-Ebene können als Instanzen der übergeordneten M3-Ebene angesehen werden. Die Elemente auf M2 und M3 haben wieder die gleiche Beziehungsart wie Elemente auf M0 und M1 oder Elemente auf M1 und M2. Dabei stellen Elemente auf M3 Kategorisierungen von Elementen auf M2 dar. M3 soll allgemein als Grundlage für die Erstellung von Metamodellen (Modellierungssprachen) dienen. Deshalb wird diese Ebene auch Metametamodellebene genannt. Die OMG setzt die M3-Ebene mit MOF gleich und dadurch sind alle OMG-Sprachtechnologien Instanzen von MOF.

Theoretisch könnten noch weitere Ebenen über M3 bzw. MOF eingeführt werden. Doch wie vorhin schon beschrieben, wird diese unendliche Problemverschiebung mit einer reflexiven M3-Ebene vermieden. Außerdem würde eine weitere Ebene keinen Abstraktionsgewinn für die praktische Modellierung bezwecken.

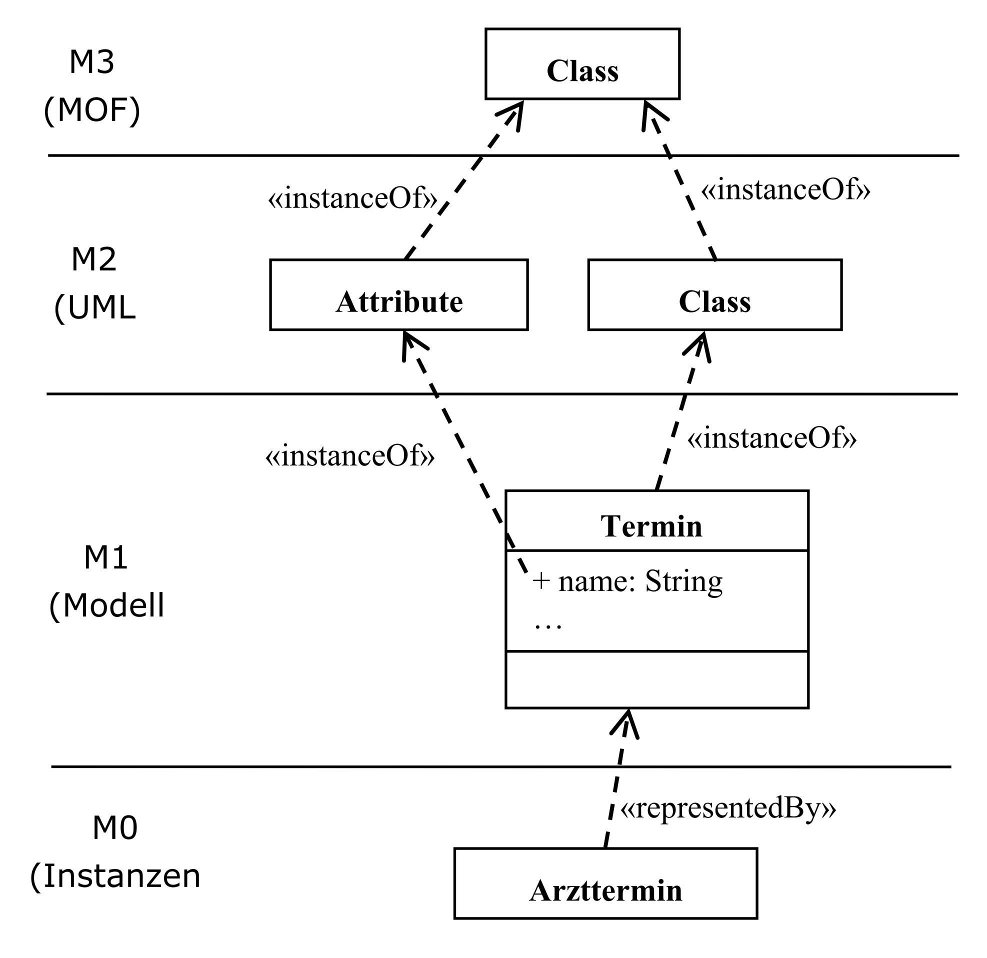


Abbildung 2.5: Metamodellhierarchie der OMG

2.3 MDA-Konzepte

In diesem Unterkapitel werden die Hauptkonzepte der Model Driven Architecture wie *Computation Independent Model*, *Platform Independent Model*, *Platform Specific Model* und *Platform Model* erläutert. Weiters wird ein Grundmechanismus der MDA, nämlich die Modelltransformationen, besprochen. Abschließend wird das Zusammenspiel der Modellarten und der Transformationen anhand eines kleinen Beispieles skizziert.

Computation Independent Model (CIM)

Ein CIM spezifiziert die Anforderungen der zu realisierenden Applikation ohne auf die rechnerunterstützte Verarbeitung einzugehen. Somit ist ein CIM unabhängig von der Realisierung des Systems, denn in dieser Phase ist noch nicht geklärt, ob überhaupt ein Computer zur Informationsverarbeitung eingesetzt wird. CIMs haben die Aufgabe ein System mit seiner Umgebung zu beschreiben und sollen ein besseres Verständnis der Aufgaben des Systems ermöglichen. Weiters wird ein CIM auch als

Domänenmodell oder Geschäftsmodell bezeichnet, denn es definiert auch ein gemeinsames Vokabular des Problembereiches. Als Notationsmöglichkeiten können beliebige Flussdiagramme – besonders geeignet sind UML2-Aktivitätsdiagramme – oder Klassen- bzw. Anwendungsfalldiagramme eingesetzt werden.

Platform Independent Model (PIM)

Ein PIM ist ein abstraktes und von jeder Implementierungstechnologie unabhängiges Modell. Im Unterschied zu CIM wird beim PIM auf Computer unterstützende Verarbeitung zurückgegriffen. Somit definiert ein PIM ein Softwaresystem, welches die Anforderungen und Funktionalität des CIM berücksichtigt. Da keine plattformspezifischen Details im PIM vorkommen, kann dieses auf verschiedenen Plattformen realisiert werden und ist somit resistent gegen Technologieveränderungen.

Platform Model (PM)

Um den Begriff Plattformmodell beschreiben zu können, muss zuerst der Begriff Plattform erklärt werden. In [OMG03a] wird eine Plattform als Menge von Technologien beschrieben, die Funktionalitäten durch Schnittstellen und Verwendungsmuster festlegt. Basiert eine Applikation auf einer Plattform, kann die Applikation auf die gesamte Funktionalität der Plattform zurückgreifen, ohne auf die Details der Realisierung der Funktionalität der Plattform eingehen zu müssen. Grundsätzlich können Plattformen grob in drei Gruppen eingeteilt werden:

- *Generische Plattformen:* Als Beispiele können Objekte oder Komponenten angegeben werden. Eine Plattform ist eine generische Objektplattform, wenn sie Konzepte wie Objekte, Vererbung und Schnittstellen anbietet.

- *Technologische Plattformen:* Als Beispiele sind CORBA als Plattform für verteilte und möglicherweise heterogene Systeme und Java 2 Enterprise Edition als Plattform für komponentenorientierte Softwareentwicklung zu nennen.

- *Herstellerspezifische Plattformen:* Diese Plattformen basieren nicht auf allgemeinen Industriestandards, haben sich jedoch durch starke Verbreitung und häufige Nutzung etabliert. Beispiele hierfür sind BEA WebLogic Server und Microsofts .NET.

Ein Plattformmodell definiert die technischen Konzepte und Elemente einer bestimmten Plattform. Weiters stellt es die Elemente, die für die Modellierung eines plattformspezifischen Systems benötigt werden, zur Verfügung.

Platform Specific Model (PSM)

Ein PSM beschreibt das System, das bereits durch CIM und PIM modelliert wurde. Der Unterschied liegt darin, dass nun plattformspezifische Details ins Modell aufgenommen werden. Dadurch enthält ein PSM Modellierungselemente, die für eine spezielle Plattform definiert wurden, bzw. in einem PM. Daher ist ein PSM auch nicht mehr plattformunabhängig.

Um die Modellarten von MDA nochmals zu verdeutlichen und den Einsatz von automatisierten Transformationen zu motivieren, wird im Folgenden ein konkretes, jedoch relativ einfaches Beispiel, das in der Abbildung 2.6 dargestellt wird, angeführt. Der Ausgangspunkt ist ein PIM mit zwei Klassen, nämlich Benutzer und Termin. Beide Klassen repräsentieren persistente Objekte und sind deshalb mit dem Stereotype *«entity»* ausgezeichnet. Zusätzlich besitzen beide Klassen jeweils ein eindeutiges Attribut zur Identifikation, welche durch *{id}* ausgezeichnet sind. Es sei darauf hingewiesen, dass keine plattformspezifischen Details in diesem Modell eingetragen sind. Durch eine Modelltransformation kann aus diesem PIM ein auf J2EE-basierendes PSM abgeleitet werden. Die beiden Klassen aus dem PIM werden in EJB Entity Beans transformiert und weitere plattformspezifische Details wie Primärschlüsseldefinitionen werden erstellt. Natürlich kann das erhaltene PSM durch eine Modell-zu-Code Transformation in Code übersetzt werden, da im plattformspezifischen Modell genügend Informationen zur automatischen Codegenerierung enthalten sind.

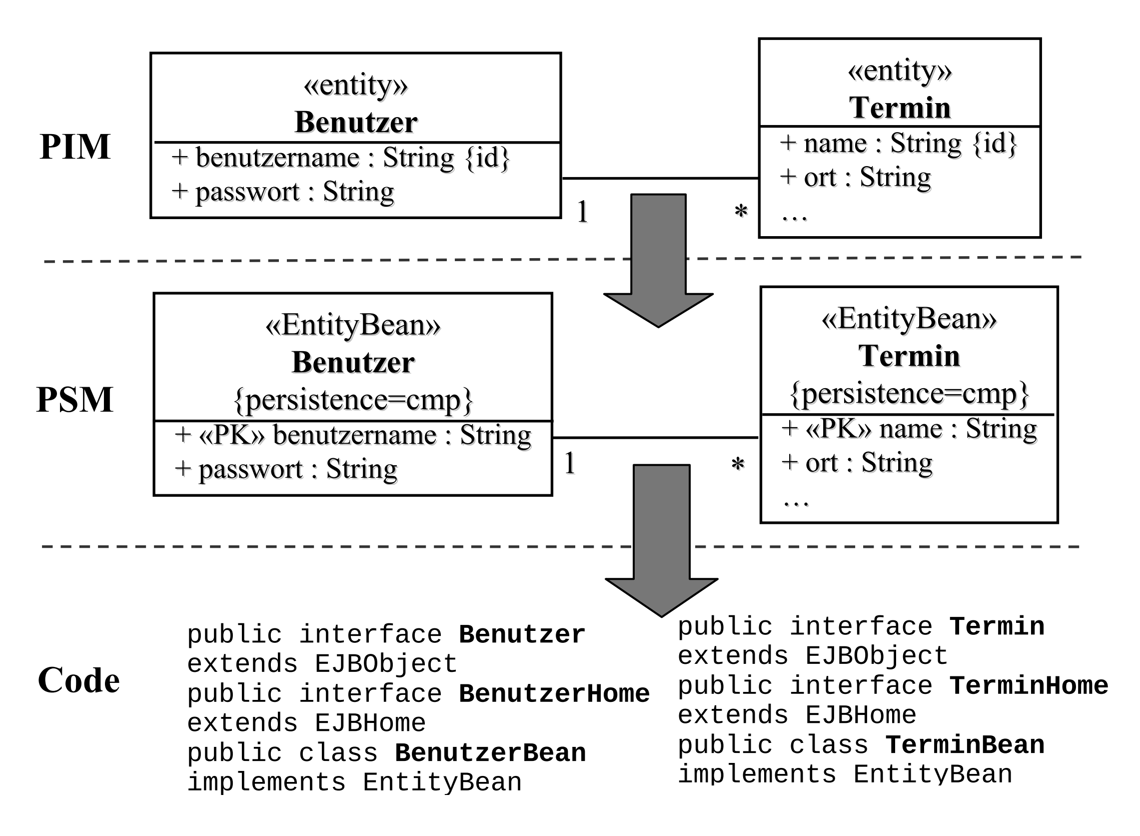


Abbildung 2.6: Beispielhaftes MDA-Projekt

Modelltransformationen

Allgemein kann eine Modelltransformation als Prozess der Umwandlung eines Modells M1 (Quellmodell), das ein System S1 beschreibt, in ein anderes Modell M2 (Zielmodell), das ebenso ein System S1 beschreibt, durch eine Transformation T1 definiert werden. Abbildung 2.7 illustriert diese allgemeine Definition einer Modelltransformation.

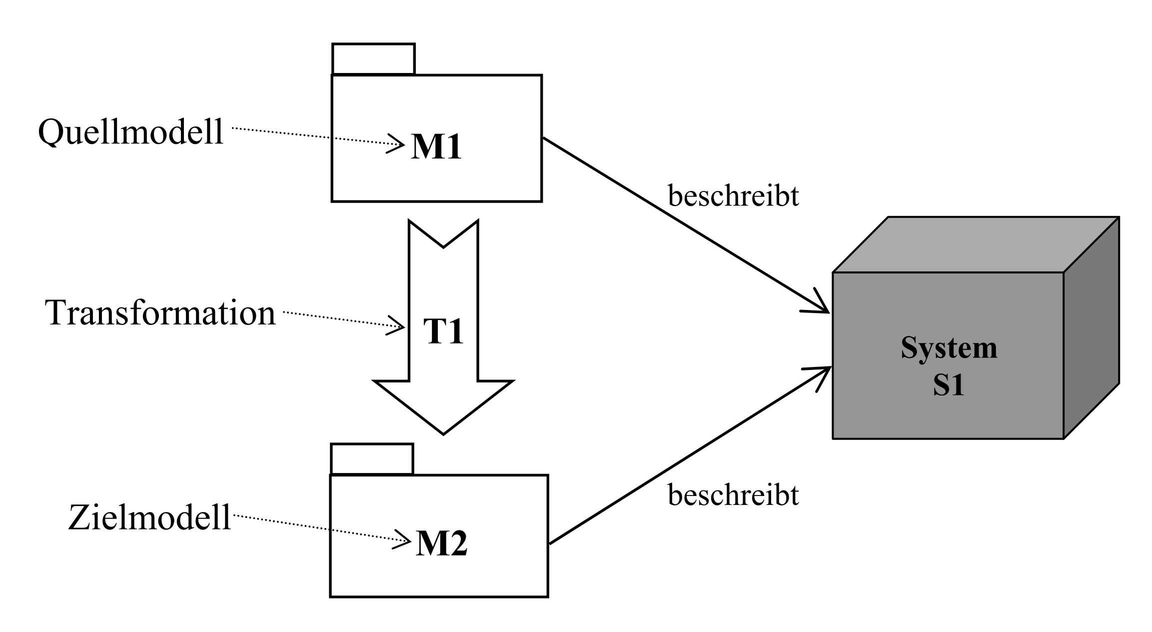


Abbildung 2.7: Definition von Modelltransformationen

Mittels Transformationen sollen neue Informationen in das Modell aufgenommen werden, wobei die eigentliche Semantik nicht verloren gehen darf. Weiters müssen auch Designentscheidungen bei Modelltransformationen berücksichtigt werden. Da man in MDA von einem sehr abstrakten und wenig detaillierten Modell (PIM) ausgeht und dieses in ein konkretes und detaillierteres Modell (PSM) transformiert, ist es mitunter sinnvoll mehrere Transformationsschritte und temporäre Zwischenmodelle einzuführen. Somit können die Aufgabenbereiche der Transformationen besser getrennt und die Komplexität der einzelnen Transformationen verringert werden. Abbildung 2.8 beschreibt einen mehrstufigen Transformationsprozess.

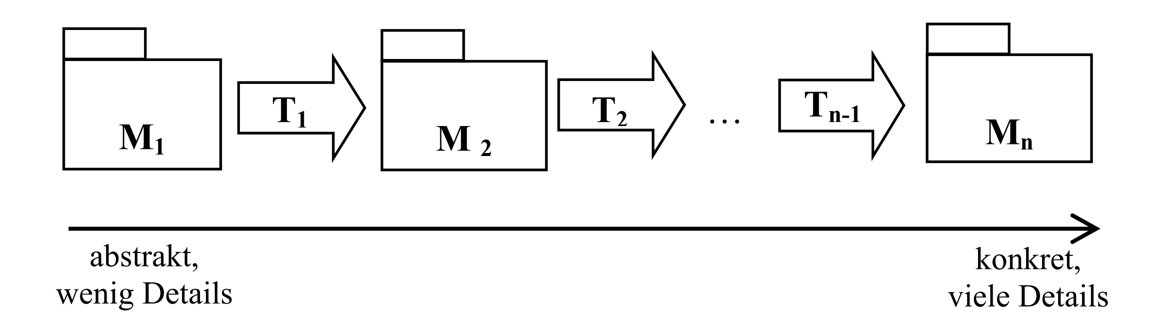


Abbildung 2.8: mehrstufiger Transformationsprozess

Der Hauptanwendungsfall von Transformationen innerhalb der MDA stellt vermutlich die Transformation eines PIMs in ein PSM dar. Für diesen Anwendungsfall werden Modelltransformationsregeln auf das PIM angewandt, um ein gewünschtes PSM zu bekommen. Um ein fertiges Softwaresystem zu erhalten, wird aber eine zweite Transformation notwendig, nämlich die Transformation des PSMs zu Softwarearte-

fakten. Generell können zwei Transformationsarten unterscheiden werden, nämlich Modell-zu-Modell Transformationen und Modell-zu-Code Transformationen.

Die Spezifikation welches Element bzw. welche Elemente des Zielmodells ein Element des Quellmodells repräsentiert bzw. repräsentieren, wird in [OMG03a] Kapitel 3.4 Abbildung genannt. Es gibt verschiedene Ansätze, welche Informationen für Abbildungen herangezogen werden. Zusammenfassend können grob drei Ansätze unterschieden werden, nämlich *Typbasierte Abbildungen*, *Instanzbasierte Abbildungen* und *Musterbasierte Abbildungen*.

Typbasierte Abbildungen

Eine typbasierte Abbildung definiert seine Abbildungsregeln mit Typen des Quell- und Zielmodells (d.h. auf Metamodellebene). Eine Regel spezifiziert, wie ein bestimmter Typ des Quellmodells auf einen Typ oder mehrere Typen des Zielmodells abgebildet wird. Metamodellabbildungen sind eine spezielle Form von typbasierten Abbildungen, wobei die Typen der Modellelemente durch das MOF-Modell spezifiziert werden sollten. Somit reicht es aus, die Transformationsregeln auf Metamodellebene zu definieren. Die Modelle auf M1-Ebene brauchen in die Transformationsdefinition nicht einzufließen, da die Regeln auf die Typen der Elemente der Modelle angewendet werden.

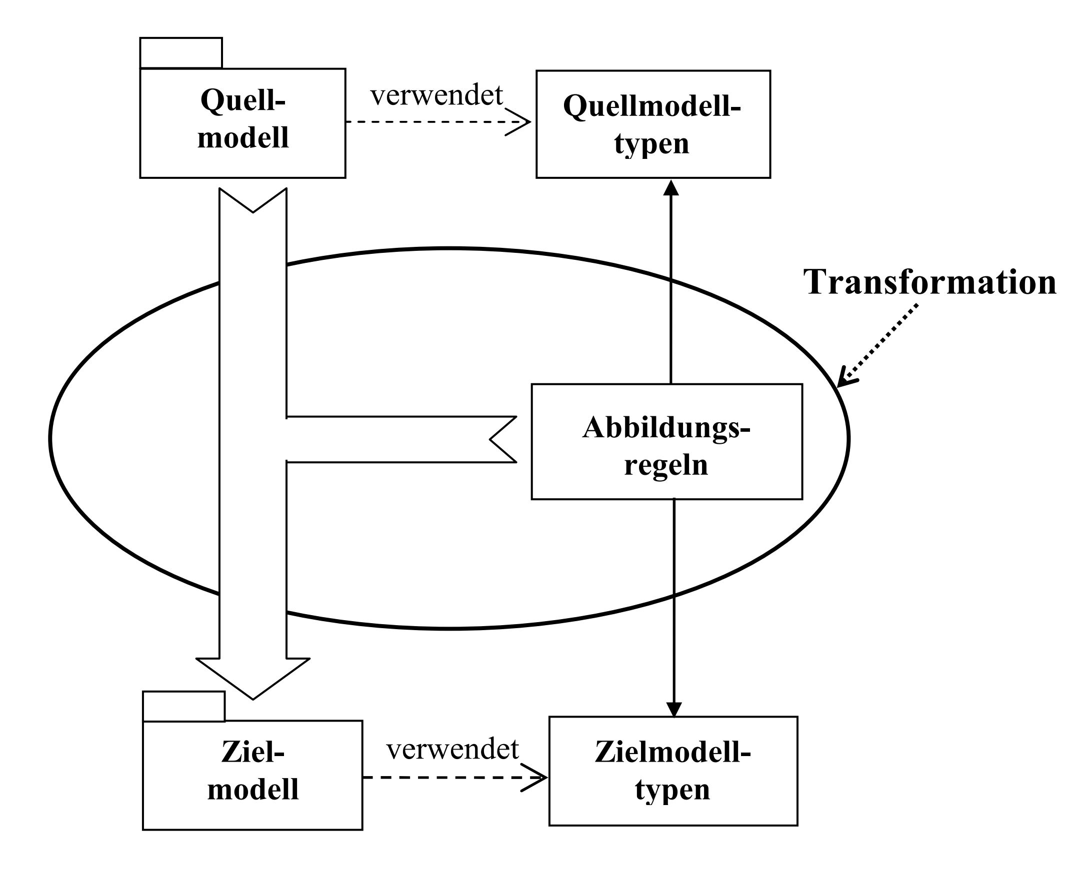


Abbildung 2.9: Typbasierte Abbildungen nach [MDA03] Kapitel 3.10.1

Instanzbasierte Abbildungen

Eine weitere Abbildungsmöglichkeit stellen instanzbasierte Abbildungen dar. Dafür werden Modellelemente im Quellmodell (d.h. auf Modellebene) ausgewählt, die auf spezielle Konzepte der Zielplattform abgebildet werden. Für diese Auswahl müssen die Elemente im Quellmodell sozusagen markiert werden.

Eine Markierung repräsentiert ein plattformspezifisches Konzept und kann zu den Elementen im Quellmodell annotiert werden, um zu beschreiben, wie diese Elemente übersetzt werden. Markierungen beinhalten somit plattformspezifische Details, die in einem plattformunabhängigen Modell nicht vorkommen sollten. Deshalb werden die Markierungen nicht direkt in einem Modell eingetragen, sondern in einer so genannten transparenten Schicht über dem plattformunabhängigen Modell annotiert.

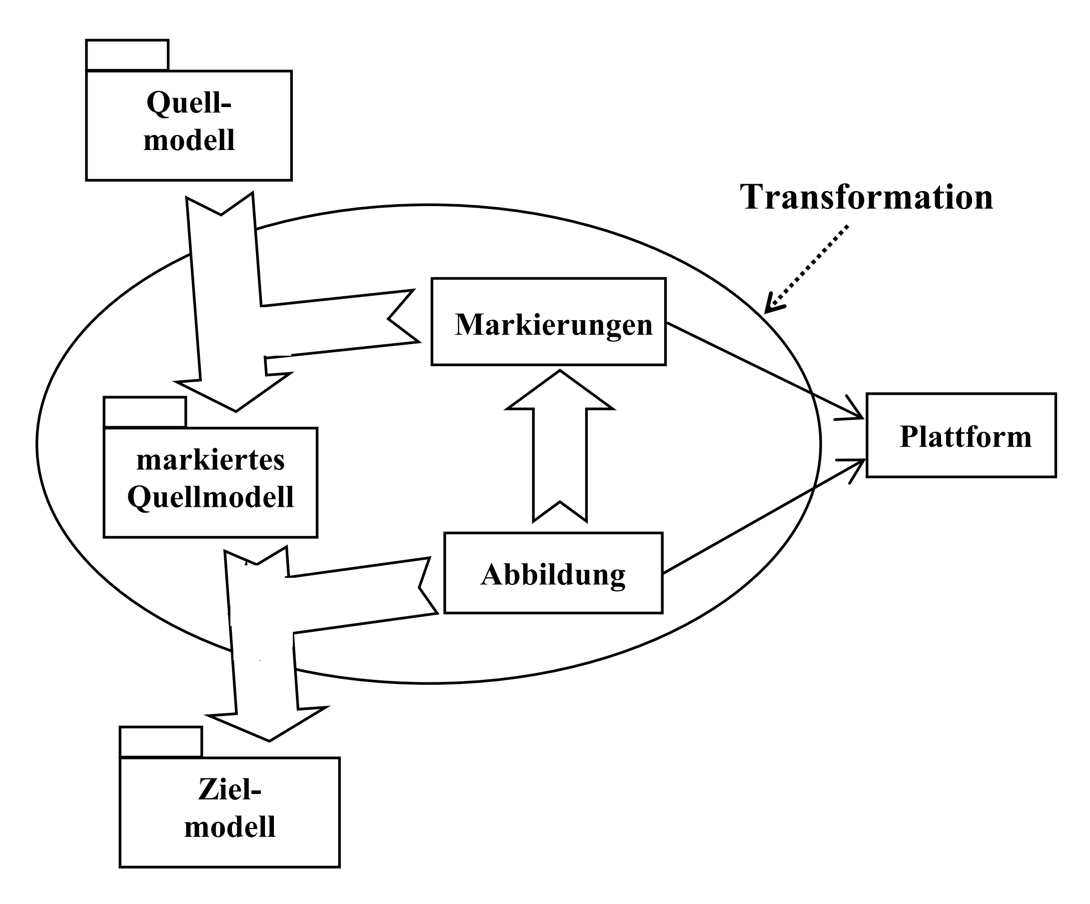


Abbildung 2.10: Instanzbasierte Abbildungen nach [MDA03] Kapitel 10.3.3

Modellelementtypen des Quellmodells besitzen eine Menge von anwendbaren Markierungen. Für die korrekte Verwendung von Markierungen, sollten diese strukturiert vorliegen, d.h. eine Menge von Markierungen, die für alternative Abbildungen eines bestimmten Konzeptes einer Plattform stehen, sollten der ArchitektIn in einem Auswahlmenü zur Verfügung stehen. Damit soll verhindert werden, dass ungültige oder widersprüchliche Markierungen an Modellelementen angebracht werden. Potentielle Markierungen für Modellelemente können in verschiedenen Bereichen gefunden werden. Hauptsächlich werden Stereotype und Schlüsselwort/Wert-Paare als Markierungen für Modellelemente eingesetzt.

In der Praxis wird eine Kombination aus typbasierten und instanzbasierten Abbildungen für den erfolgreichen Einsatz von Transformationen erwartet. Eine typbasierte Abbildung bildet einen Typ des Quellmodells in einem Typ oder in mehrere Typen des Zielmodells ab. Ist nun keine Möglichkeit für das Anbringen von Markierungen vorgesehen, werden nur Informationen der Metamodelle des Quell- und Zielmodells für die Generierung des Zielmodells verwendet. Doch oft bedarf es an Markierungen im Quellmodell, damit das Zielmodell die geforderten funktionalen und nicht funktionalen Eigenschaften erfüllt, die aber aus dem Quellmodell nicht direkt abgeleitet werden können. Viele der heute erhältlichen MDA-

Entwicklungswerkzeuge setzen eine Kombination aus typbasierten und instanzbasierten Abbildungen ein (siehe ArcStyler, Objecteering UML, OptimalJ in Kapitel 5). Diese Werkzeuge ziehen für Typbasierte Abbildungen Stereotypen und UML Metaklassen und für Instanzbasierte Abbildungen Schlüsselwort/Wert-Paare heran.

Musterbasierte Abbildungen

Für öfters vorkommende Anordnungen von Modellelementen im Quellmodell, welche auf bestimmte Anordnungen von Modellelementen im Zielmodell abgebildet werden, können sogenannte musterbasierte Abbildungen eingesetzt werden. Dabei definieren Transformationsregeln typbasierte Abbildungen von Modellelementmuster des Quellmodells in Modellelementmuster eines Zielmodells. Zusätzlich können noch Markierungen auf Instanzebene angebracht werden, um die Entwurfsoptionen festzulegen. Musterbasierte Abbildungen sind mit Entwurfsmuster vergleichbar, jedoch enthalten erstgenannte noch zusätzliche Informationen über den Transformationsablauf wie z.B. die Reihenfolge der angewandten Transformationsschritte.

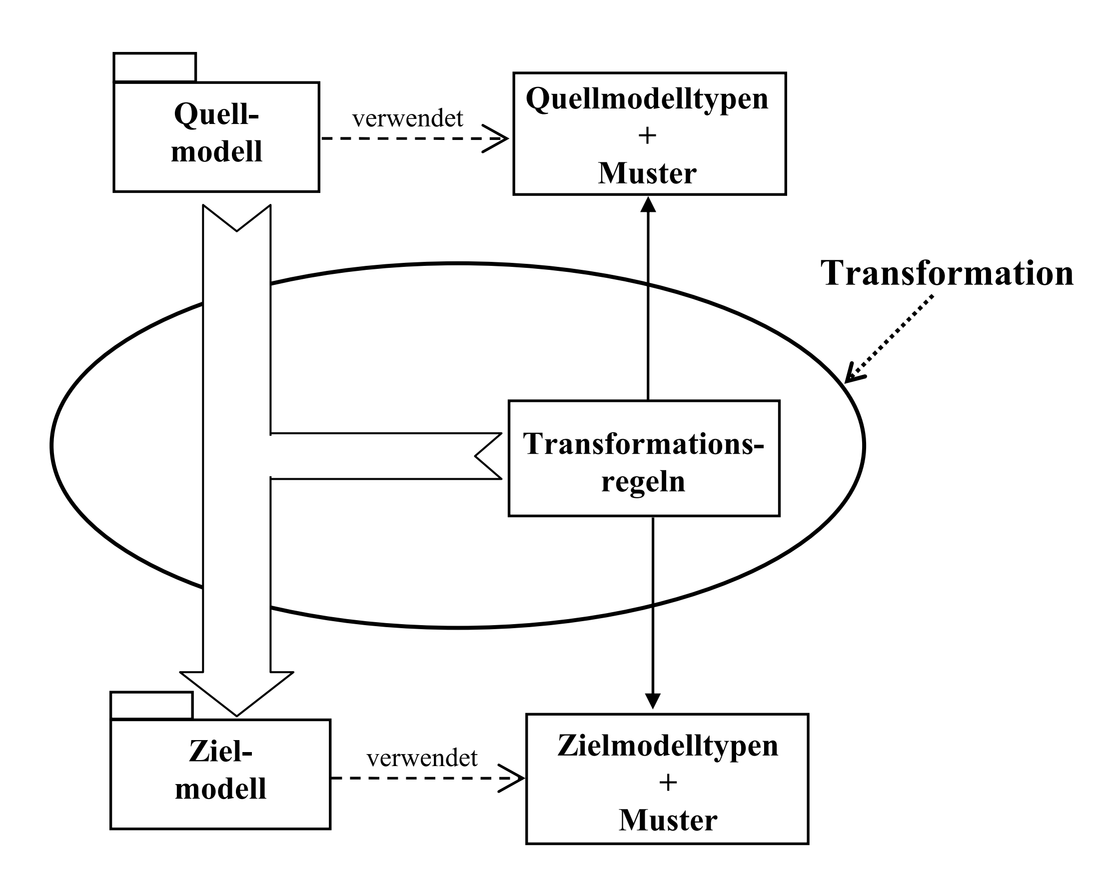


Abbildung 2.11: Musterbasierte Abbildungen nach [MDA03] Kapitel 10.3.4

Musterbasierte Abbildungen werden zurzeit in keinem kommerziellen MDA-Entwicklungswerkzeug eingesetzt, jedoch beschäftigen sich einige Forschungspro-

jekte mit der Erstellung von auf musterbasierten Abbildungen spezialisierten Transformationswerkzeugen. Realisiert werden die musterbasierten Transformationen durch Graphentransformationsregeln. Als Beispiele für graphenbasierte Transformationswerkzeuge siehe GreAT [Chri04], UMLX [Will03] und BOTL [Mars03]. Eine ausgezeichnete Einführung in Graphtransformationen für Modelle findet man in [Heck00] und [Bare02].

Transformationen sollten nur bedingt manuell ablaufen, im Allgemeinen aber automatisiert durchgeführt werden, da durch Automatisierung eine enorme Produktivitätssteigerung erreicht wird. Um die Investitionen in Transformationendefinitionen zu schützen, bedarf es einer standardisierten Transformationssprache. Die OMG erkannte diese Notwendigkeit und leitete die Standardisierung einer MOF-basierenden Query/View/Transformation Sprache ein. Leider konnte bis heute noch kein Standard durchgesetzt werden, da sich fünf der acht eingereichten Vorschlägen noch im Rennen um die Standardisierung befinden. Die Anforderungen an eine Query/View/Transformation Sprache und die Realisierung dieser Anforderungen in den eingereichten Vorschlägen werden in [Gard03] und [Lang03] besprochen.

Kapitel 3

MDA-Werkzeuge

In diesem Kapitel wird auf Werkzeuge, welche eine sinnvolle Verwendung in einem MDA-Projekt finden können, näher eingegangen. Da die Vorteile des modellgetriebenen Software Engineerings erst mit dem Einsatz von Werkzeugen realisiert werden, ist es eine Notwendigkeit, die passenden Werkzeuge für einen konkreten Anwendungsfall zu finden. Am Softwaremarkt werden zahlreiche MDA-kompatible Tools angeboten, aber nur ein kleiner Teil der Produkte wird seinen Ansprüchen der Erfüllung des MDA-Paradigmas gerecht. Da MDA heutzutage einen lukrativen Marketingfaktor darstellt, werden leider auch einige ältere Tools, wie plattformspezifische Skeleton-Codegeneratoren und CASE Tools, unter dem Namen MDA neu vermarktet. Diese Entwicklung birgt aber auch eine potentielle Gefahr für die MDA. Durch unausgereifte oder neu vermarktete Produkte, die den Ansprüchen der ProgrammierInnen oder ProjektleiterInnen nicht gerecht werden, kann die MDA-Entwicklung leicht zum Stocken gebracht oder vollständig gestoppt werden. Deshalb wird in diesem Kapitel der MDA-Werkzeugsektor eingehend besprochen, um eine Vorstellung von gut realisierten Werkzeugen zu geben.

Das Unterkapitel 3.1 beschäftigt sich mit einem möglichen Entwicklungsprozess für den Einsatz des MDA-Paradigmas. Im Abschnitt 3.2 werden die Anforderungen an MDA-Werkzeuge im Laufe eines Entwicklungsprozesses in einem Kriterienkatalog gesammelt. Danach wird im Unterkapitel 3.3 eine mögliche Kategorisierung von MDA Tools diskutiert. Weiters wird ein grober Überblick über die am Markt angebotenen Produkte gegeben, um eine erste Entscheidungsgrundlage zu bieten, ob ein Tool für ein konkretes Projekt sinnvoll einzusetzen ist.

3.1 MDA-Entwicklungsprozess

Um die Anforderungen an MDA-Werkzeuge zu definieren, müssen wir uns zuerst die Aktivitäten eines Softwareentwicklungsprozesses mit MDA-Ansatz ansehen. Anhand der Aktivitäten wird ersichtlich, welche Tätigkeiten in einem MDA-Projekt auftauchen, und entscheidbar, welche Tätigkeiten durch Werkzeugeinsatz automatisiert werden können. Im Folgenden wird eine mögliche Variante eines MDA-Entwicklungsprozesses beschrieben, wobei auf mögliche Iterationen nicht eingegangen wird, da primär nur die Aktivitäten und nicht der Ablauf des Prozesses für die Kriterien von MDA-Werkzeugen von Interesse sind. Abbildung 3.1 skizziert den vorgeschlagenen MDA-Entwicklungsprozess.

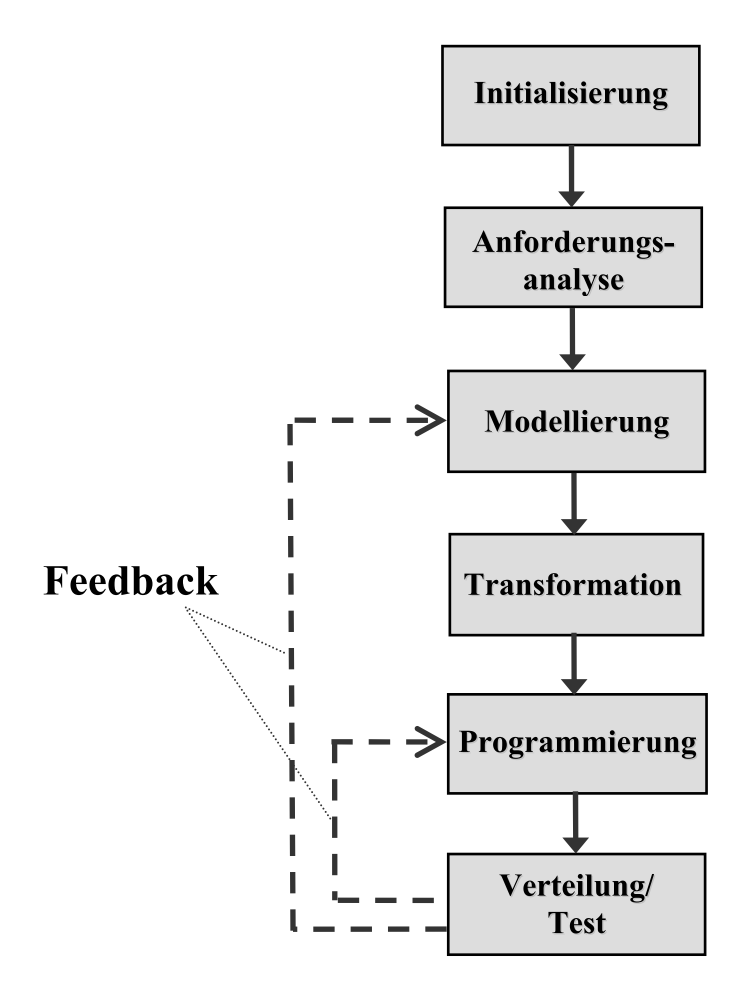


Abbildung 3.1: möglicher MDA-Entwicklungsprozess

Initialisierung

Natürlich startet ein MDA-Projekt, wie jedes andere Softwareentwicklungsprojekt auch, mit einer Projektinitialisierungsphase. Dabei werden meist Projektmanagementtätigkeiten, wie Mitarbeitereinteilung, Zeitplanentwurf, einzusetzende Techno-

logien und vieles mehr, durchgeführt. In dieser Phase wird der Einsatz eines MDA-Werkzeuges zwar noch keinen Sinn ergeben, aber diese Phase wird die Entscheidung, welches MDA-Werkzeug sinnvoll eingesetzt werden kann, stark beeinflussen, da jedes MDA-Werkzeug andere Fähigkeiten besitzt und unterschiedliche Plattformen unterstützt. Weiters muss in dieser Phase die Modellierungssprachen, die Plattformen des Quell- und Zielmodells und die Transformationen identifiziert werden.

Anforderungsanalyse

Mit den ersten Überlegungen bezüglich der Anforderungen der zu erstellenden Software wird der Einsatz eines MDA-Werkzeuges notwendig. Die Anforderungen sollten als Anwendungsfälle modelliert werden und einen ersten Startpunkt der modellgetriebenen Softwareerstellung darstellen. Dabei sollte sich die Modellierung auf einer abstrakten und berechnungsunabhängigen Ebene befinden.

Modellierung

MDA rückt Modelle in das Zentrum des Entwicklungsprozesses und dadurch stellt die Modellierung des Softwaresystems eines der Hauptaufgaben des Entwicklungsprozesses dar. Ein Modell sollte alle strukturellen und dynamischen Eigenschaften eines Systems spezifizieren, um diese Definitionen durch Transformationen zu verfeinern und auf einer Plattform abzubilden. Sind keine Transformationen für die Verfeinerung des Quellmodells vorhanden, müssen diese erstellt werden. Idealerweise sollten die Transformationen modelliert werden und nicht nur in Codeform vorliegen. Somit ist diese Phase in zwei Bereiche aufgeteilt: einerseits in die Modellierung des zu erstellenden Systems und andererseits in die Modellierung der benötigten Transformationen. Bevor ein Modell durch eine Transformation verfeinert wird, sollte das Modell auf die Realisierung der in der Anforderungsphase definierten Anforderungen durch eine Modellvalidierung überprüft werden. Zusätzlich sollte das Modell durch eine Modellverifikation auf formale Fehler untersucht werden.

Transformation

Die durch die EntwicklerInnen erstellten Modelle müssen weiterverarbeitet werden, um plattformspezifische Details einzubringen und schlussendlich den Code für die Anwendung zu generieren. Dafür müssen in dieser Phase zwei Transformationsarten angewandt werden. Einerseits sollten Modell-zu-Modell Transformationen für die automatische Erstellung von konkreteren und detailreicheren Modellen durchgeführt

werden und andererseits Modell-zu-Code Transformationen für die Erstellung der Artefakte angewandt werden.

Programmierung

Der von MDA-Werkzeugen automatisch generierte Code stellt im Allgemeinen noch keine einsatzfähige Anwendung dar. Die EntwicklerInnen müssen noch die Geschäftslogikmethoden ausprogrammieren, da die meisten MDA-Werkzeuge keine automatische Generierung von Geschäftslogik anbieten. In dieser Phase müssen auch Testskripte erstellt bzw. ergänzt werden, da diese durch die heutigen Technologien nicht komplett aus den Modellen abgeleitet werden.

Verteilung/Test

Die letzte Phase des Softwareentwicklungsprozesses stellt die Verteilung und den Test der Applikation dar. Die einsatzfähigen Komponenten müssen auf verschiedene Maschinen verteilt und konfiguriert werden. Möglicherweise müssen die neuen Komponenten in vorhandene Systeme (*Legacy*-Systeme) integriert werden. Die Funktionalität der neuen Komponenten und das Zusammenspiel mit vorhandenen Systemen muss schließlich noch durch Tests überprüft werden.
MDA bietet auch für die Verteilung und für das Testen der Applikation Unterstützung für die EntwicklerInnen. Auf Modellebene können Testfälle oder sogar vollständige Testsuiten spezifiziert werden und aus diesen kann der benötigte Code für die Durchführung der Tests abgeleitet werden. Um die Verteilung der Komponenten zu beschleunigen bzw. zu automatisieren, können Skripte, die die Verteilung der Komponenten bzw. Initialisierung der Testdaten vornehmen, aus den Modellen abgeleitet werden.

3.2 Kriterienkatalog für MDA-Werkzeuge

Eines der Hauptforschungsziele dieser Diplomarbeit ist die Erstellung eines Kriterienkataloges für die Evaluierung von MDA Tools. Anhand des Kriterienkataloges soll der Erfüllungsgrad der MDA-Unterstützung eines Tools, welches von sich behauptet, das MDA-Paradigma umzusetzen, bestimmbar sein.

Um einen strukturierten Überblick über die Anforderungen zu ermöglichen, werden diese in Kategorien eingeteilt. Im Folgenden werden die Kategorien identifiziert und danach ihre Anforderungen besprochen.

Im ersten Teil dieses Abschnitts wurden mögliche Aktivitäten eines Entwicklungsprozesses mit MDA-Ansatz beschrieben. Betrachtet man den vorgeschlagenen Entwicklungsprozess, so sind daraus drei Kategorien, welche besonders von der Automatisierung durch Tooleinsatz profitieren, ableitbar.

- *Modellerstellung*: Dieser Bereich beschäftigt sich mit Anforderungen an die grafische Modellierung der Softwaresysteme, aber auch nichtgrafische Elemente wie Einschränkungen oder plattformspezifische Markierungen werden hier berücksichtigt.

- *Modelltransformation*: Diese Kategorie evaluiert die Unterstützung von Modelltransformationen. Der Bereich Modelltransformation sollte von MDA-Werkzeugen stark unterstützt werden, da dieser Bereich den Grundgedanken der MDA darstellt.

- *Artefakterzeugung*: Hier werden Anforderungen an den Artefaktgenerierungsprozess berücksichtigt. Um ein einsatzfähiges Softwaresystem zu erhalten, müssen Artefakte aus den Modellen erzeugt werden. Dies sollte großteils automatisch durch Werkzeugunterstützung erfolgen.

Die durch Prozessaktivitäten definierten Kategorien werden durch zwei weitere ergänzt. Diese sind nicht unmittelbar aus den einzelnen Aktivitäten des Entwicklungsprozesses ableitbar, aber dennoch für die Evaluierung von großem Interesse.

- *Toolinteroperabilität/Toolintegration*: Diese Anforderungskategorie beschäftigt sich mit der Nutzung von zukünftigen oder bereits etablierten Tools (wie z.B. Eclipse oder NetBeans) im Entwicklungsprozess. Hierfür kann es sich als nützlich erweisen, wohldefinierte Modellaustauschformate oder Schnittstellen für eine mögliche Integration von Tools in einem MDA-Werkzeug anzubieten.

- *Ökonomische Kriterien*: Die fünfte und letzte Kategorie untersucht die wirtschaftlichen Kriterien eines MDA-Tooleinsatzes. Eine sinnvolle bzw. berechtigte Verwendung von MDA Tools kann nur dann erfolgen, wenn die laufenden Einsparungen die Kosten der Entwicklungsumstellung übersteigen.

Nachdem die Kategorien definiert sind, wird mit den einzelnen Kategorien und der Definition ihrer zugehörigen Anforderungen fortgefahren. Jede Anforderung erhält

ein eindeutiges Kürzel, welches sich aus einem Buchstaben und einer laufenden Nummer je Kategorie zusammensetzt. Zusätzlich wird dem Kürzel eine sprechende Kurzbezeichnung beigefügt. Anhand dieser Beschriftung wird im Kapitel 4.7 auf die hier folgenden definierten Anforderungen referenziert, um eine übersichtliche Werkzeugevaluierung zu gewährleisten.

3.2.1 Modellerstellungskriterien

Diese Kategorie beschäftigt sich mit Anforderungen, welche für die Erstellung von rechnerverarbeitungsfähigen Modellen erfüllt sein sollten, um einen positiven Nutzen aus dem MDA-Paradigma zu ziehen.

Um Modelle für Modelltransformation und/oder Codegenerierung einsetzen zu können, müssen diese Modelle zuerst existieren bzw. erstellt werden. Dafür sollte ein MDA-Werkzeug alle Modellierungsaktivitäten in einem Entwicklungsprozess abdecken, denn je mehr Informationen in den Modellen enthalten sind, desto präziser sind die Transformationsregeln anwendbar. Weiters sind für die korrekte Modellerstellung Simulationen und Validierungen wichtige Bestandteile und sollten bei der Erstellung jederzeit anwendbar sein. Im Folgenden werden die Anforderungen dieser Kategorie aufgelistet und beschrieben.

M1: Unterstützung von UML 1.x / 2.0

UML wird als Kernmodellierungssprache für die modellgetriebene Softwareentwicklung gesehen, da UML zurzeit die meiste Verwendung in Softwareprojekten findet. EntwicklerInnen und ArchitektInnen sind mit UML vertraut und haben diese Sprache bereits in bisherigen Softwareprojekten für Dokumentationen und Anforderungsanalysen eingesetzt.

Um dem Anspruch an MDA-Unterstützung gerecht zu werden, sollte ein MDA Tool standardmäßig die Erstellung von UML-Diagrammen unterstützen, d.h. es sollte der AnwenderIn das gesamte Spektrum von Struktur- und Verhaltensdiagrammen zur Verfügung stehen. Weiters sollten die Notationselemente mit UML standardkonformer Syntax im Tool angeboten werden. Leider war in bisherigen UML CASE Tools die Syntax der Notationselemente leicht verändert oder es gab zusätzliche, nicht UML-konforme Notationselemente bzw. wurden einige Standardnotationselemente überhaupt nicht unterstützt.

Mit der Entstehung des MDA-Paradigmas wurde das Bedürfnis nach einer ausdrucksstärkeren und auf Metaebene bzw. Metametaebene konsistenteren Modellierungssprache als UML 1.x erkannt. Die OMG reagierte mit der Standardisierung einer neuen UML-Version (UML 2.0), die zurzeit als OMG Adopted Version [OMG03f] vorliegt. Da der Standard noch nicht in der finalen Version festgelegt wurde, warten noch viele Toolanbieter mit der Implementierung der Version 2.0 in ihrem Produkt ab.
Hersteller von MDA-Werkzeugen sollten UML 2.0 so rasch wie möglich in ihre Produkte integrieren, da UML 2.0 das Potential der modellgetriebenen Softwareentwicklung um einiges steigert. Gerade bei den Verhaltensdiagrammen werden in der Version 2.0 deutlich mehr Modellierungsmöglichkeiten geboten, aber auch bei den Strukturdiagrammen, wie z.B. den Komponentendiagrammen, werden für MDA nützliche Konzepte integriert.

M2: Erstellung von UML Profiles

Um PIMs und PSMs zu erstellen, ist es notwendig, plattformspezifische Informationen, seien es abstrakte oder konkrete Plattformen, in die Modelle einzuarbeiten. Die OMG stellt dafür UML Profiles für verschiedenste Bereiche und Plattformen zur Verfügung. Toolhersteller sollten für die aktuellsten Plattformen Profiles bereitstellen und es den BenutzerInnen ermöglichen, eigene Profiles zu erstellen bzw. die bereitgestellten Profiles zu modifizieren. Dabei sollten Profiles aus Stereotypen, Schlüsselwort/Wert-Paaren und Einschränkungen bestehen. Zusätzlich sollte ein integrierter Profile-Editor angeboten werden, der die Bearbeitung bzw. Erstellung von Profiles erleichtert.

M3: Definition von auf MOF basierenden Metamodellen

UML Profiles sind eine mögliche Erweiterungsform des UML-Metamodells. Oft reicht aber eine Erweiterung nicht aus und es ist die Modellierung eines neuen Metamodells notwendig. Um der Forderung der OMG, MOF als Metametamodellierungssprache zu benutzen, nachzugehen, sollten Toolhersteller die Nutzung von MOF zur Erstellung neuer Metamodelle erlauben und weiters auch in ihren Tools unterstützen. Dabei sollte in den Tools ein MOF-Modellierungseditor angeboten werden, der die korrekte Erstellung eines auf MOF basierenden Metamodells erleichtert.

Neue Metamodelle erlauben nicht nur die Unterstützung zukünftiger und domänenspezifischer Modellierungstechniken, sondern auch die Nutzung von älteren Paradigmen.

M4: Repository für (Meta-)Modelle

Ein Repository („Modelldatenbank“) ist eine möglicherweise verteilte Komponente, in der Modelle gespeichert werden und die Zugriff auf diese Modelle mittels wohldefinierter Schnittstellen ermöglicht. Diese Schnittstellen können entweder durch standardisierte oder auch durch proprietäre APIs implementiert werden. Eine weitere Variante stellen Abfragesprachen dar, die Modellelemente durch Queries liefern können.

MDA-Werkzeuge sollten mit einem eigenen Repository für die persistente Speicherung und das Ein- bzw. Auslesen von Modellen bzw. Metamodellen ausgestattet sein. Die Verwendung von auf MOF basierenden Repositories ermöglicht den Zugriff auf (Meta-)Modelle, die von anderen Werkzeugen erstellt wurden. Weiters stellt die Verwendung eines Repositories die Grundlage für die Durchführung eines vernünftigen Versionsmanagements und Mehrbenutzerzugriffs dar.

M5: OCL-Unterstützung

MDA-Werkzeuge sollten Unterstützung für den Einsatz von OCL als Einschränkungssprache anbieten. Ein weiteres nützliches Feature wäre ein integrierter OCL-Editor, der automatische Syntaxüberprüfungen durchführt. Die im Modell angegebenen Einschränkungen sollten in die Transformationen, speziell in die Codegenerierung, einfließen. Mittels Modellsimulationen sollte eine Überprüfungsmöglichkeit von Einschränkungen geboten werden, um bereits in frühen Phasen Fehler in Modellen zu finden und zu eliminieren.

M6: Modellverifikation

Fehler in Softwaresystemen sollten im Lebenszyklus eines Softwaresystems so früh wie möglich erkannt werden. Idealerweise unterstützt das MDA-Paradigma Fehlererkennung bereits auf Modellebene. Somit werden Probleme rechtzeitig erkannt und können kostengünstiger behoben werden als bei „normalen“ Entwicklungsprozessen.

Ein sehr effektives Konzept zur frühen Fehlererkennung stellt die Modellvalidierung dar. MDA-Werkzeuge sollten für ihre bereitgestellten Plattformen auch Verifikationsmöglichkeiten berücksichtigen, um der AnwenderIn Feedback über die Korrektheit des konstruierten Modells zu geben. Zusätzlich sollte das Tool über Definitionsmöglichkeiten für eigene Verifikationsregeln verfügen, die manuell oder automatisch ausgeführt werden.

M7: Modellsimulation

Um Modelle zu simulieren, muss zuerst das Verhalten des Systems im Modell festgelegt werden. Deshalb sollten MDA-Werkzeuge Spezifikationsmöglichkeiten für das Verhalten eines Systems bieten, um Simulationen auf Modellebene durchführen zu können. Durch Simulationen auf Modellebene können Fehler lange vor der Codegenerierung und Verteilung der Softwarekomponenten gefunden werden. Meistens wird das Verhalten durch Action Semantic Sprachen und/oder Zustandsdiagrammen angegeben. Simulationsdurchläufe sollten mit verschiedenen Startzuständen und Szenarien (z.B. mit unterschiedlichen Signalfolgen) simulierbar und protokollierbar sein.

M8: Unterstützung von Markierungen

Bereits in [OMG03a] werden Markierungen als notwendiges Konzept für die Auswahl von Transformationsmöglichkeiten eines Modellelements beschrieben. MDA-Werkzeuge sollten Notationsmöglichkeiten für Markierungen an Modellelementen implementieren. Der AnwenderIn sollten intelligente Dialoge für die Vergabe von Markierungen geboten werden, um die korrekte Verwendung von Markierungsvarianten zu ermöglichen und um unsinnige Markierungen von Elementen zu vermeiden. Die Markierungen müssen bei den Transformationen automatisch einfließen.

Es sollte auch möglich sein, Markierungen für mehrere unterschiedliche Plattformen parallel zu vergeben, z.B. für EJB und gleichzeitig für .NET. Die angewandten Transformationen müssen selber eruieren können, welche Markierungen für sie relevant sind, und nicht relevante Markierungen von anderen Plattformen ignorieren.

M9: User Interface Modellierung

Für heutige Applikationen stellt ein ausgereiftes User Interface ein notwendiges Feature dar. Daher sollten MDA-Werkzeuge eine Möglichkeit bieten, Benutzerschnitt-

stellen zu modellieren oder zumindest in einer proprietären Form zu definieren. Eine abstrakte Spezifikation der Benutzerschnittstelle durch Modelle würde die automatische Erstellung von mehreren Varianten von User Interfaces Implementierungen ermöglichen. Somit wären JSPs für ein Webinterface oder Swing-Klassen für eine Applikation aus dem Modell ableitbar.

Standardmäßig bietet UML keine Modellierungsmöglichkeit von User Interfaces, aber UML Profiles für User Interface Definitionen könnten in naher Zukunft realisiert werden. Bis dahin könnten Tools spezielle Metamodelle für den User Interface Entwurf verwenden oder den Zugriff auf die modellierten Komponenten aus externen User Interface Erstellungstools, wie z.B. MacroMedia DreamWeaver, ermöglichen.

M10: Unterstützung von abstrakten/konkreten Plattformen

Damit plattformunabhängig modelliert werden kann, sollten abstrakte Plattformkonzepte, wie z.B. Objekte oder Komponenten, standardmäßig unterstützt werden. Dies wird bereits hauptsächlich durch UML 1.x bzw. UML 2.0 Diagrammarten abgedeckt.

Um die effiziente Erstellung von Softwaresystemen zu fördern, sollten MDA Tools die aktuellsten Technologien wie EJB, .Net oder CORBA unterstützen, um die Entwicklung von plattformspezifischen Modellen oder die Markierung von plattformunabhängigen Modellen zu ermöglichen. Daher sollten Datentypen, Stereotype und Schlüsselwort/Wert-Paare für die aktuellsten Plattformen in den Tools verfügbar sein.

M11: Unterstützung von Action Semantic

Für Zustands- und Aktivitätsdiagramme sind die Notationen von Aktionen besonders interessant, um das Verhalten präziser zu spezifizieren und so durch Simulationen zu überprüfen. UML 1.4 besitzt keine Aktionen, somit reduzieren sich die Simulationsmöglichkeiten, da keine konkreten Aktionen beobachtbar sind.

MDA Tools sollten daher die Möglichkeit bieten, Aktionen zu Zustands- und Aktivitätsdiagrammen beizufügen, um Simulationen und umfangreichere Codegenerierung zu ermöglichen. Da für Action Semantics keine standardisierte Sprache existiert, muss in der Praxis auf eine der vielen Sprachvarianten (ASL [KC01] oder OAL

[AT04]) zurückgegriffen werden. Natürlich wäre eine automatische Syntaxüberprüfung zweckmäßig, um keine fehlerhaften Modellspezifikationen auf Codeebene zu übertragen.

M12: Anforderungsanalyse

Definition und Management von Anforderungen, wie z.B. Nummerierung der Anwendungsfälle, Versionierung der Anwendungsfälle oder Dokumentation, sollten von MDA Tools angeboten werden. Eine weitere wichtige Funktion stellt die Traceability von Modellelementen (auch auf Codeebene) zu den Anforderungen dar und vice versa. Die Anwendungsfälle sollten am Beginn des Entwicklungsprozesses als erster Anknüpfungspunkt für weitere Modellierungsschritte fungieren.

M13: Modellierung von Testfällen

Da im MDA-Paradigma Modelle die Basis für den generierten Code darstellen, ist es nützlich, die Testfälle aus den Modellen abzuleiten und diese ebenso zu modellieren. Dadurch wird auch die automatische Codegenerierung für Testfälle möglich. Ein MDA Tool sollte somit die Definition und das Management von einzelnen Testfällen bzw. von gesamten Testsuiten unterstützen.

Es existieren mehrere Metamodelle für die Modellierung von Testspezifikationen wie z.B. *Testing and Test Control Notation* (*TTCN*) [ITU03]. Da diese aber nicht auf MOF basieren und ihre eigenen Diagrammnotationen verwenden, werden eigene Tools für ihren Einsatz benötigt. Eine Lösung wären Integrationsmöglichkeiten in einem MDA-Werkzeug oder der Aufbau eines MDA Frameworks, das die Nutzung der auf Tests spezialisierten Tools ermöglicht. UML Profiles für Testspezifikationen könnten weitere Erleichterungen für die Testerstellung und Verwaltung mit sich bringen.

M14: Mehrbenutzerfähigkeit

Abgesehen von kleinen Projekten mit geringem Personaleinsatz stellt Modellierung, wie die Softwareentwicklung im Allgemeinen, eine Teamaktivität dar. Deshalb sollten MDA-Werkzeuge mehrbenutzerfähig ausgerichtet sein. Gleichzeitiger Mehrbenutzerzugriff auf gleiche Modellteile und Versionierung sind nur einige wichtige Anforderungen an die Modelldatenbank. Da die Hauptinformationen nicht wie bei

herkömmlichen Softwareprojekten in Dateien vorliegen, sondern in Modelldatenbanken gespeichert sind, erschwert sich die Erfüllung dieser Anforderung.

Für größere Projekte mit vielen MitarbeiterInnen wird ein verteilter Modelldatenbankserver notwendig, wodurch mehrere EntwicklerInnen gleichzeitig auf verschiedene Modellelemente zugreifen und diese bearbeiten können. Haben mehrere MitarbeiterInnen auf die gleichen Modellelemente Zugriff, sollte das Tool einen eindeutigen Authentifikationsmechanismus besitzen, um protokollieren zu können, wer welche Änderungen am Modell durchgeführt hat.

3.2.2 Modelltransformationskriterien

Diese Kategorie beschäftigt sich ausführlich mit den Anforderungen an die Modelltransformationen. Modelltransformationen strukturieren den Artefaktgenerierungsprozess in mehrere Stufen. Ohne Modelltransformationen würden alle notwendigen Transformationsschritte zugleich im Codegenerierungsprozess spezifiziert werden, der in diesem Fall zu einem äußerst komplexen und dadurch auch schwer verwaltbaren Prozess anwachsen würde. Im Folgenden werden die Anforderungen an MDA Tools der Kategorie Modelltransformation aufgelistet und beschrieben.

T1: Modell-zu-Modell Transformation

[OMG03a] definiert mehrere Transformationsarten im Zuge eines Projekts mit MDA-Ansatz, wie beispielsweise die Transformation von PIM zu PSM oder vice versa. Verfügt ein MDA-Werkzeug über unterschiedliche Modellebenen, so sollten auch automatische Transformationen zwischen diesen angeboten werden. Bei der automatischen Erstellung bereits vorhandener Modelle muss darauf geachtet werden, dass manuelle Veränderungen im Modell erhalten bleiben, d.h. die manuellen Änderungen sollten nicht durch neue Transformationen überschrieben werden.

T2: Modell-zu-Code Transformation

Ein Hauptziel der MDA ist die automatische Generierung von Code aus Modellen. Daher sollte ein MDA Tool eine Funktion für die Generierung von Artefakten aus Modellelementen anbieten. Dabei sollte nicht nur Programmcode erzeugt werden, sondern auch alle für die erfolgreiche Inbetriebnahme der Software notwendigen Dateien. Man denke dabei an XML-Dateien wie Deployment Descriptors oder an SQL-Skripte.

T3: Traceability zwischen Quell- und Zielelementen

Der Zusammenhang von Modellelementen auf unterschiedlichen Modellebenen und daraus resultierenden Codefragmenten wird ohne Zusatzinformationen auf den ersten Blick schwer erkennbar sein. Deshalb sollten Tools die Transformationsabhängigkeiten auf unterschiedlichen Ebenen durch zusätzliche Informationen kennzeichnen. Ein weiteres Feature wäre eine Log-Datei für den Transformationsprozess, in der mitprotokolliert wird, welches Quellelement in welches Zielelement transformiert wurde.

T4: Modifikation/Erstellung von Transformationen

Um für zukünftige Technologieentwicklungen gewappnet zu sein, sollten MDA Tools die Erstellung von selbstdefinierten Transformationen anbieten. Durch Modifikationsmöglichkeiten von vorhandenen Transformationen könnte die Berücksichtigung von neuen Versionen von Standards oder Plattformen erreicht werden. Eine Grundvoraussetzung dafür ist aber, dass die Quelldateien der Transformationsspezifikationen vom Hersteller mitgeliefert werden.

T5: Debugging Informationen

Um einen Einblick in einen konkreten, ausgeführten Transformationsablauf zu gewähren, sollte ein Tool Debugging Informationen für Transformationen anzeigen. Dieses Feature stellt gerade bei fehlerhaften Transformationen eine große Hilfestellung dar, da man einen ersten Anhaltspunkt bekommt, an welcher Stelle der Fehler in einer komplexen Transformation liegen könnte.

T6: Modellierung von Transformationen

Um die Erstellung von Transformationsdefinitionen zu erleichtern, sollte eine Modellierungsmöglichkeit für Modell- und Codetransformationen durch MDA Tools geboten werden. Aus diesen Modellen sollte der Code für die Transformationen automatisch erzeugt werden, da die Transformationserstellung als Spezialfall der modellgetriebenen Softwareentwicklung gesehen werden kann. Zusätzlich sollten Transformationen das Vererbungskonzept, das aus objektorientierten Sprachen bekannt ist, unterstützen. Somit können beispielsweise Funktionalitäten für Java-Transformationen auch für EJB-Transformationen wiederverwendet werden. Durch die Modellierung von Transformationen sind die technischen Architekturen von

Plattformen, verglichen mit textuellen Transformationsspezifikationen, einfacher zu verstehen.

3.2.3 Artefaktgenerierungskriterien

Um aus konkreten und detaillierten Modellen lauffähige Softwaresysteme zu generieren, wird eine letzte Transformation notwendig, nämlich eine Modell-zu-Code Transformation. Theoretisch kann Code auch als Modell betrachtet werden, und daher eine Modell-zu-Code Transformation als gewöhnliche Modelltransformation gesehen werden. Doch in der Praxis zeigen sich spezielle Anforderungen an die Artefaktgenerierung, wie beispielsweise die Berücksichtigung von manuellen Änderungen in textbasierten Artefakten. Im Folgenden werden die Anforderungen der Kategorie Artefaktgenerierung aufgelistet und beschrieben.

A1: Geschützte Bereiche

Da heutzutage eine 100%ige Codegenerierung meist nicht erreicht wird, muss ein Großteil der Geschäftslogik manuell hinzugefügt werden. Um manuelle Änderungen im generierten Code nicht beim nächsten Generierungszyklus zu verlieren, sollten MDA Tools Schutzmechanismen für manuell erstellte Codefragmente anbieten. Je ausgereifter dieser Schutzmechanismus implementiert ist, desto weniger manuelle Nacharbeiten werden notwendig.

Geschützte Bereiche sollten von MDA Tools unbedingt unterstützt werden, und dies nicht nur bei Programmcodedateien wie etwa Java-Klassen, sondern auch bei allen sonstigen Textdateien wie z.B. SQL-Skripten oder XML-Dateien.

A2: Debugging Informationen für Artefaktgenerierung

Wie schon bei der Anforderung T5: Debugging Informationen für Modelltransformationen besprochen wurde, sollten MDA Tools für die Artefaktgenerierung ebenso Debugging Informationen bieten. Gerade für das Testen von neuen Generierungsspezifikationen bietet dieses Feature einen großen Nutzen für die rasche Fehlerfindung.

A3: Hohe Codequalität

Da meistens keine 100%ige Codegenerierung durchführbar ist, müssen EntwicklerInnen Codefragmente zu generierten Codeteilen manuell hinzufügen. Hierfür ist es von besonderer Bedeutung, dass MDA Tools den generierten Code gut strukturieren und die Codeteile durch Kommentare beschreiben. Hohe Quelltextqualität trägt einiges zur Produktivitätssteigerung beim manuellen Hinzufügen von Codefragmenten bei.

A4: Automatische Dokumentationserstellung

Ein ausgereiftes MDA-Werkzeug sollte zumindest eine Möglichkeit der automatischen Erstellung einer Dokumentation, welche sowohl grafische Diagramme als auch im Modell enthaltene Textfragmente umfasst, bieten. Flexible Dokumentationsgeneratoren sollten veränderbare Templates anbieten, um individuelle Dokumentationsanforderungen zu verwirklichen.

MDA Tools sollten auch die Generierung von HTML-Dokumentationen für Modelle anbieten. HTML-Dokumentationen bieten eine statische Sicht auf die Modellelemente. Weiters haben sie den Vorteil, dass sie rasch über einen Browser angezeigt werden können ohne ein Modellierungstool zu verwenden. In der erstellten Dokumentation sollte für jedes Diagramm eine entsprechende Grafik enthalten sein. Eine Navigation durch das gesamte Modell ist über Hyperlinks einfach zu realisieren.

3.2.4 Toolinteroperabilität/Toolintegration

Diese Kategorie listet Anforderungen an die Erweiterung von MDA Tools bzw. die Integration von für den Entwicklungsprozess nützlichen Tools, wie grafische Modellierungseditoren oder Programmier-IDEs, auf. Da sich am UML-Toolsektor und am IDE-Toolsektor bereits mehrere ausgereifte Tools etabliert haben und EntwicklerInnen diese Produkte seit mehreren Jahren verwenden, sollte eine Möglichkeit geboten werden, auf diese Tools auch im MDA-Entwicklungsprozess zurückzugreifen. Grafische Modellierungseditoren könnten entweder über wohldefinierte Schnittstellen in das MDA Tool integriert werden oder die Modelle könnten über das XMI-Format zwischen den Tools ausgetauscht werden. Für die Integration von Programmier-IDEs ist heutzutage die automatische Erstellung von Projektdateien für die jeweilige IDE ausreichend, da diese noch keinen MDA Ansatz unterstützen und deshalb nur für die Formulierung der nicht automatisch erzeugbaren Geschäftslogik auf Codeebene verwendet werden.

Ein weiterer interessanter und in Zukunft immer wichtiger werdender Aspekt ist die Integration und Nutzung von Open Source IDE-Plattformen wie Eclipse[1] und Net-Beans[2]. NetBeans verfügt bereits über ein auf MOF-basierendes Repository namens Meta Data Repository (MDR). Für Eclipse werden erweiterbare Plugins für Modellierung, wie das Eclipse Modeling Framework, angeboten. Es ist bereits absehbar, dass in Zukunft laufend neue MDA-Aspekte zu diesen Plattformen hinzugefügt werden. Im Folgenden werden die Anforderungen der Kategorie Toolinteroperabilität und Toolintegration aufgelistet und beschrieben.

I1: Unterstützung von XMI

Um die Unabhängigkeit der Modelle von Tools zu bewahren und nicht in Zukunft in einer speziellen Technologie gefangen zu sein, muss eine Möglichkeit geboten werden, um die erstellten Modelle für andere Tools nutzbar zu machen. MDA Tools bieten unterschiedliche Funktionalitäten. Daher ist eine Kette von Tools durchaus sinnvoll, um die erfolgreiche und effiziente Entwicklung eines Softwaresystems zu gewährleisten.

Zumindest der Import und Export von Modellen über das XMI Format sollte ein MDA Tool anbieten, um die Modelle mit anderen Tools zu teilen. Ein auf MOF basierender XMI-Austausch wäre noch vorteilhafter, da er den Export von Modellen als MOF-Instanzen anstatt Instanzen des UML-Metamodells ermöglicht.

Die Unterscheidung zwischen Modellimport/-export und Codegenerierung durch Transformationen ist in den meisten Tools nicht immer transparent. Dennoch würde eine Modelltransformation nach XMI mehr Flexibilität mit sich bringen, denn dadurch würde die Modifikation des XMI-Formats durch die AnwenderIn erlaubt. Da oft unterschiedliche Dialekte von XMI in Modellierungstools eingesetzt werden, würde dieses Feature der BenutzerIn die Möglichkeit bieten, die unterschiedlichen XMI-Formate abzugleichen.

Um die Wiederverwendung von vorhandenen Modellen zu erhöhen, sollte der Import/Export von einzelnen Modellteilen unterstützt werden. Meistens wird nicht ein komplettes Modell wiederverwendet, sondern nur ein bestimmter Ausschnitt. Ein Tool sollte z.B. den Import/Export von einzelnen Packages oder Komponenten er-

---

[1] www.eclipse.org (Stand: 1.1.2005)

[2] www.netbeans.org (Stand: 1.1.2005)

möglichen. Dafür ist aber die Grundvoraussetzung, dass Modelle in mehrere Bereiche (Domänen) unterteilt werden und keine monolithischen Modelle erstellt werden.

I2: Erweiterung

In Zukunft werden kontinuierlich neue Anforderungen an MDA Tools hinzukommen. Darum sollte ein Tool weitere Komponenten aufnehmen können, um auf neue Entwicklungen reagieren zu können. Zumindest sollten neue Modellierungstools integrierbar sein, um bei der Erstellung von Modellen auf individuelle Bedürfnisse eingehen zu können.

Frameworks wie Eclipse und NetBeans sind Referenztechnologien für Erweiterungsmöglichkeiten und Toolintegration, doch leider bieten diese Technologien heutzutage noch zu wenig Modellierungsaspekte, um MDA zu unterstützen.

I3: Unterstützung von Programmier-IDEs

MDA Tools sollten die automatische Erstellung von Projektdateien für die meistverwendeten IDEs, wie JBuilder, Eclipse und NetBeans als Beispiele im Javabereich, unterstützen. Da bei großen Projekten eine Vielzahl von Dateien und Verzeichnissen durch Artefaktgenerierung entsteht, hilft dieses Feature der EntwicklerIn, Zeit zu sparen und sich mehr auf ihre Programmiertätigkeiten zu konzentrieren.

3.2.5 Ökonomische Kriterien

Diese Kategorie beschäftigt sich mit den wirtschaftlichen Anforderungen an MDA Tools. Wie bei jeder Entscheidung über den Einsatz neuer Technologien sind auch bei MDA Tools Kosten-Nutzen-Rechnungen angebracht.

Natürlich sind die Anschaffungskosten eines Tools eine wichtige Entscheidungsgrundlage, ob dieses Tool eingesetzt werden soll. Doch noch viel kritischer sind die laufenden Kosten, die zum Großteil durch Personalkosten verursacht werden. Deshalb ist sicherzustellen, dass einerseits die EntwicklerInnen das neue Tool akzeptieren und andererseits rasch lernen, damit umzugehen.

In der Regel wird am Beginn des Einsatzes der neuen Technologien die Produktivität kurzfristig sinken. Erst nach einer gewissen Einarbeitungszeit kann die Produktivität über das Produktivitätslevel der älteren Technologie gesteigert werden. Je kürzer die

Zeitspanne der Einarbeitungsphase ist, desto schneller amortisieren sich die Investitionen. Ausführliche Dokumentationen und angebotene Schulungen sind maßgeblich daran beteiligt, die Einarbeitungszeit zu reduzieren. Im Folgenden werden die Anforderungen der Kategorie ökonomische Kriterien aufgelistet und näher beleuchtet.

O1: Anschaffungskosten

Die Anschaffungskosten eines Tools sind, verglichen mit den laufenden Kosten, relativ einfach zu bestimmen. Gerade im MDA-Toolsektor werden einige Open Source Produkte kostenlos angeboten, dagegen veranschlagen kommerzielle Toolanbieter oft hohe Anschaffungspreise. Dieses Kriterium soll dem Leser einen Preisvergleich zwischen den Tools ermöglichen und die Spanne zwischen den minimalen und maximalen Anschaffungskosten von kommerziellen Tools aufzeigen.

O2: Unterschiedliche Versionen

Um kundenspezifische Ansprüche an MDA Tools zu berücksichtigen, sollten vom Toolhersteller verschiedene Versionen angeboten werden, welche sich vom Funktionsumfang und daher auch preislich unterscheiden.

Während einige Software EntwicklerInnen nur Codeskelette aus Modellen ableiten und den Rest manuell programmieren, haben sich andere EntwicklerInnen auf Transformationen spezialisiert und brauchen dafür erweiterte Toolfunktionen. Deshalb sollten zumindest für die Entwicklung von Softwaresystemen und für die Erstellung von Metamodellen und Transformationen zwei verschiedene Editionen angeboten werden. Ein Tool, das alle Rollen eines Softwareentwicklungsprozesses unterstützt, wirkt schnell überladen und besitzt dadurch unzureichende Benutzerfreundlichkeit.

O3: Angebotene Schulungen

Wie bei der Einleitung dieser Kategorie erwähnt, müssen die BenutzerInnen lernen, mit dem Tool umzugehen. Dies kann durch angebotene Schulungen, entweder direkt beim Toolhersteller oder bei speziellen Beratungsunternehmen, erleichtert werden. Auch für die Akzeptanz des Tools bei den neuen BenutzerInnen ist eine gute Einschulung förderlich.

O4: Werkzeugdokumentationen

Wohl eine der wichtigsten Anforderungen stellt die Forderung an guter und ausreichender Dokumentation des Werkzeuges dar. Dabei soll die interne Struktur des Tools gut beschrieben sein und zusätzlich sollten Tutorials für die wichtigsten Projekte verfügbar sein. Weiters sollten Beispielanwendungen mit der Distribution mitgeliefert werden, um den EntwicklerInnen eine erste Orientierungshilfe bei der Erstellung eigener Anwendungen zu geben.

Speziell für die Erstellung bzw. Modifikation von Metamodellen und Transformationen sind Dokumentationen und ausführliche Referenzbeispiele unbedingt notwendig. Denn ohne diese Hilfen scheint eine erfolgreiche Erweiterung des Tools aussichtslos.

O5: Supportmöglichkeit

Findet man keine Hilfestellung für ein konkretes Problem im Benutzerhandbuch oder weist das Tool interne Fehler auf, sollte Hilfe über eine Supportmöglichkeit verfügbar sein. Bei kommerziellen Tools könnte eine Supportmöglichkeit über einen Helpdesk ablaufen oder über persönliche Betreuer vor Ort. Bei Open Source Produkten werden sich die Supportmöglichkeiten auf Mailinglisten und Benutzerforen beschränken.

O6: Benutzerfreundlichkeit

Natürlich trägt die Benutzerfreundlichkeit eines Tools sehr zu seiner Akzeptanz bei den AnwenderInnen und seiner Produktivität bei. Die Anforderung Benutzerfreundlichkeit ist im Vergleich zu den bisherigen am schwierigsten objektiv zu bewerten. Um einer Scheinobjektivität entgegen zu wirken, wird eine subjektive Bewertung durchgeführt, die sich großteils auf folgende Kriterien stützt:

- Die Steuerung der MDA-Aktivitäten über eine grafische Benutzeroberfläche wird höher bewertet als eine Steuerung über die Kommandozeile.

- Falls ein Tool MDA-Aktivitäten gesammelt in einer integrierten Umgebung bewerkstelligt, wird es höher bewertet als ein Tool, das diese Aktivitäten nur durch das Hin- und Herwechseln zwischen verschiedenen Tools realisiert.

3.3 Kategorisierung von MDA-Werkzeuge

Dieses Unterkapitel soll dem Leser eine Orientierung am MDA-Werkzeugmarkt geben. Zur Zeit wird eine Vielzahl an verschiedensten Tools am MDA-Marktsektor angeboten, jedoch unterscheidet sich der Funktionsumfang der Tools stark voneinander. Um trotzdem einen guten Überblick geben zu können, werden in diesem Abschnitt die angebotenen MDA-Werkzeuge in vier Kategorien eingeteilt: Executable UML Tools, technologiespezifische Tools, eigentliche MDA Tools und Codegeneratoren.

Executable UML Tools existieren schon seit einigen Jahren und wurden schon öfters in der Industrie erfolgreich eingesetzt. Die Tools versuchen, die Konzepte der Executable UML für die Entwicklung von Software zu implementieren. Nähere Informationen über Executable UML werden im Unterkapitel 3.3.1 gegeben.
Technologiespezifische Tools (siehe Unterkapitel 3.3.2) ermöglichen die Entwicklung von Softwaresystemen durch den Einsatz von MDA-Konzepten. Werkzeuge dieser Kategorie unterstützen aber nur eine bestimmte Plattform, wie J2EE und .NET.
Eigentliche MDA Tools (siehe Unterkapitel 3.3.3) implementieren einen Großteil der von MDA geforderten Konzepte und sind auch nicht auf bestimmte Technologien beschränkt. Weiters können diese Tools durch ArchitektInnen für jegliche Plattformen erweitert werden.
Herkömmliche *Codegeneratoren* (siehe Unterkapitel 3.3.4) dürfen nicht mit MDA Tools verwechselt werden, da diese meistens keine MDA-Konzepte implementieren. Werkzeuge dieser Kategorie sind nur auf die Produktion von Skeleton-Code spezialisiert und auch nicht erweiterbar.

Im Folgenden werden die vier Kategorien aufgelistet, wobei für jede Kategorie die wichtigsten Vertreter angeführt werden.

3.3.1 Executable UML Werkzeuge

MDA stellt einen sehr abstrakten Lösungsvorschlag für die Entwicklung von Softwaresystemen dar. Es werden sozusagen nur die Rahmenbedingungen vorgegeben, die eigentliche Implementierung von MDA-Konzepten wird den Toolherstellern zugeteilt.

Dagegen ist Executable UML ein konkreter Vorschlag, wie MDA realisiert werden kann. Doch Executable UML verwendet nicht die gesamte Spannbreite an MDA-

Konzepten, sondern beschränkt sich auf die Ebenen PIM und Code. Da die Schöpfer von Executable UML die PSM-Ebene nur als temporäres Hilfsmodell für die Erstellung von Code aus PIM-Modellen sehen und da durch den Executable UML Ansatz der gesamte Code einer Anwendung erzeugt wird, verzichtet man auf das PSM-Konzept zur Gänze. Es reicht also für Executable UML Projekte aus, ein PIM für das Softwaresystem zu erstellen. Ein Model Compiler transformiert ein Executable UML Modell in eine konkrete Implementierung (z.B. C++) unter der Verwendung von Informationen über die Software- und Hardwarezielplattformen.

UML bietet eine einheitliche Notation für die Repräsentation von unterschiedlichen Aspekten von objektorientierten Systemen. Executable UML verwendet eine ausgewählte Teilmenge der möglichen UML-Notationen, um ausführbare Modelle zu erstellen [Rais04]. Definiert wurde Executable UML von Mellor und Balcer in [Mell02]. Ein Executable UML Modell nutzt vorwiegend Klassendiagramme, Zustandsdiagramme und Aktionen. Weitere UML Diagramme können zur Verdeutlichung von Systemfunktionalitäten eingesetzt werden. Da sie aber keine Ausführungssemantik in Executable UML besitzen, können sie weder simuliert noch transformiert werden. Anwendungsfalldiagramme und Aktivitätsdiagramme können für die Anforderungsanalyse eingesetzt werden, bevor der eigentliche Systementwurf beginnt. Im Folgenden werden die ausführbaren Konzepte – Klassendiagramm, Zustandsdiagramm und Aktion – näher erläutert. Zusätzlich bietet Tabelle 3.1 eine grobe Zusammenfassung der ausführbaren Konzepte.

- *Klassendiagramm:* Diese Diagrammart wird eingesetzt, um die formale Struktur des Systems zu zeigen, wobei alle Entitäten des Systems identifiziert und als Klassen abstrahiert werden. Wichtige Eigenschaften der Entitäten werden als Attribute modelliert und Beziehungen zwischen den Entitäten als Assoziationen. Operationen werden nicht explizit in Klassen angegeben, da Operationen in Form von Aktionen in Zustandsdiagrammen angegeben werden.
  Ein System sollte in Domänen unterteilt werden, um übersichtlichere Teilsysteme zu erhalten, wobei ein Teilsystem aus Klassen und ihren Beziehungen besteht. Jede Domäne wird für sich selbst modelliert, ohne auf andere Domänen einzugehen. Um Beziehungen zwischen Domänen zu berücksichtigen, sind sogenannte Bridges („Brücken") zwischen Domänen definierbar. Mit Hilfe der Brücken sind Informationen zwischen Domänen über genau spezifizierte Verbindungspunkte austauschbar. Die verschiedenen Domänen

und ihre Abhängigkeiten können durch Pakete und Paketabhängigkeiten visualisiert werden.

- *Zustandsdiagramm:* Jede im Klassendiagramm spezifizierte Entität besitzt einen Lebenszyklus, der durch ein Zustandsdiagramm beschrieben wird. Somit besitzt jede Klasse ein Zustandsdiagramm, um seine temporären Zustandsänderungen und sein Verhalten zu beschreiben. Ein Zustandsdiagramm formalisiert den Lebenszyklus eines Objektes anhand von Zuständen, Ereignissen und Transitionen. Ein Zustandsdiagramm kann alternativ als Zustands-Transitions-Tabelle visualisiert werden. In einer solchen Tabelle beschreibt jede Zeile einen Zustand und jede Spalte ein Ereignis. Zustandsdiagramme synchronisieren ihr Verhalten untereinander durch das Senden von Signalen, die von empfangenden Zustandsdiagrammen als Ereignis interpretiert werden. Ein Signal wird als asynchrone Nachricht, die Daten transportiert, betrachtet. Durch den Empfang eines Signals feuert eine Transition im Zustandsautomaten, wobei eine Prozedur ausgeführt wird. Diese Prozedur muss bis zu ihrem Ende durchlaufen werden, erst danach kann ein neues Signal verarbeitet werden.

- *Aktion:* Das Verhalten von Systemen ist durch die Änderungen von Objektzuständen, die aufgrund von Ereignissen geschehen, definiert. Jedes Zustandsdiagramm verfügt über eine Menge von Prozeduren, wobei genau eine bei einem Zustandsübergang ausgeführt wird. Jede Prozedur besteht wieder aus einer Menge von Aktionen. Eine Aktion wird als eine atomare Berechnung, wie eine Datenabfrage oder die Erzeugung eines Objektes, definiert und wird durch eine Action Semantic Language spezifiziert. Aktionen sind ähnlich zu Anweisungen auf Codeebene, aber mit dem Unterschied, dass sie sich auf einer höheren Abstraktionsstufe befinden und keine Annahmen über die Implementierung treffen.

Konzept	**Notationsart**	**Notationselemente**
Entitäten	Klassendiagramm	Klassen Attribute Einschränkungen
Lebenszyklen der Entitäten	Zustandsdiagramm	Zustände Ereignisse Transitionen Prozeduren
Aktionsspezifikationen	Action Semantic Language	Aktionen

Tabelle 3.1: Executable UML Konzepte nach [Mell02]

Ein Executable UML Modell enthält alle notwendigen Details, um die Ausführung, Verifikation und Validierung unabhängig von der Implementierung zu unterstützen. Es werden aber keine Entwurfs- oder Codedetails im Modell angegeben, um das Modell auszuführen. Daher können auch Testfälle anhand des Modells ausgeführt werden, um Anforderungen des Systems relativ früh im Entwicklungsprozess zu überprüfen.

Executable UML Tools brauchen besondere Funktionen, um die Aktionen eines Objektes beschreiben zu können. Dies erfordert den Einsatz einer Action Semantic Language. Die bekanntesten Tools im Executable UML Sektor sind mit einer eigenen, proprietären Action Semantic Language ausgestattet. Die Tabelle 3.2 listet die bekanntesten Executable UML Toolhersteller und ihre Tools auf.

Toolhersteller	**Produkt**	**Action Language**	**Homepage**
Kennedy Carter	iUML/iCCG	Action Specification Language (ASL)	http://www.kc.com
Accelerated Technology	Nucleus BridgePoint	Object Action Language (OAL)	http://www.acceleratedtechnology.com
Kabira Technologies	Kabira Design Center	Kabira Action Semantics (Kabira AS)	http://www.kabira.com
Pathfinder Solutions	PathMATE	PathMATE's Action Language (PAL)	http://www.pathfindermda.com

Tabelle 3.2: Executable UML Tools

Für die Toolevaluation in Kapitel 4 werden für den Bereich Executable UML die Tools iUML/iCCG und Nucleus BridgePoint genauer betrachtet. Das Tool

iUML/iCCG wurde schon für eine Vielzahl von Projekten im Embedded Systems Bereich eingesetzt und die Firma Kennedy Carter ist intensiv an der Entwicklung von UML und der Integration von Action Semantics beteiligt. Nucleus BridgePoint wird deshalb genauer betrachtet, weil Mellor und Balcer, die Hauptinitiatoren von Executable UML, bei Accelerated Technology beschäftigt sind und so das Tool die Konzepte der Executable UML wohl am vollständigsten realisieren wird.

3.3.2 Technologiespezifische Werkzeuge

Einige MDA-Werkzeuge sind auf eine spezifische Technologie ausgerichtet, wofür sie besondere Funktionalität, wie z.B. plattformspezifische Benutzeroberflächenerstellung oder automatische Generierung von Verteilungsinformationen für bestimmte Herstellerplattformen, bieten. Die EntwicklerInnen sollten sich aber im Klaren sein, dass sie für die unterstützte Plattform ausgezeichnete Hilfestellung bekommen aber gleichzeitig auch in dieser gefangen sind. Speziell für Java und .NET werden einige spezialisierte Werkzeuge angeboten. Beispielsweise unterstützt Compuwares OptimalJ die Entwicklung von Softwaresystemen, die auf J2EE basieren. Dot Net Builders Constructor fokussiert auf die Erstellung von Systemen, welche auf der .NET Plattform basieren. IBM bietet zwei verschiedene Versionen seines Rational XDE Werkzeuges an: eine Version für die Erstellung von EJB-Systemen und eine weitere für die Entwicklung in .NET. Tabelle 3.3 gibt einen groben Überblick über plattformspezifische MDA-Werkzeuge.

Toolhersteller	**Produkt**	**unterstützte Plattform**	**Homepage**
Compuware	OptimalJ	J2EE	http://www.compuware.com
Dot Net Builders	Constructor	.NET	http://www.dotnetbuilders.com
IBM	Rational XDE	J2EE, .NET	http://www-306.ibm.com/software/awdtools/developer/java (für J2EE) http://www-306.ibm.com/software/awdtools/developer/visualstudio (für .NET)
BITPlan	UML2PHP	PHP	http://www.uml2php.com
Web Models	WebRatio	J2EE, .NET	www.webratio.com

Tabelle 3.3: plattformspezifische MDA-Werkzeuge

Für die Evaluierung in Kapitel 4 wird OptimalJ als Vertreter der technologiespezifischen MDA-Werkzeuge herangezogen, da OptimalJ bereits für mehrere Projekte erfolgreich eingesetzt wurde und bei weitem das ausgereiftere Werkzeug dieser Kategorie darstellt.

3.3.3 Eigentliche MDA-Werkzeuge

Zurzeit existiert kein Werkzeug, das alle Konzepte der MDA optimal umsetzt. Doch einige MDA-Werkzeuge unterstützen durch ausgereifte Modellierungseditoren einen Großteil der geforderten Konzepte, wie z.B. die Definition von UML Profiles, Transformationen und Metamodellen.

In den letzten drei Jahren entstanden viele neue MDA-Werkzeuge und dieser Trend wird sich auch in der Zukunft aufgrund des immer populärer werdenden MDA-Paradigmas fortsetzen. Tabelle 3.4 gibt einen groben Überblick über die aktuell angebotenen MDA-Werkzeuge.

Toolhersteller	Produkt	Homepage
Interactive Objects	ArcStyler	http://www.io-software.com
SOFTEAM	Objecteering/UML	http://www.objecteering.com
AndroMDA Team (open source)	AndroMDA	http://www.andromda.org
b+m Informatik AG (open source)	b+m Generator FrameWork	http://architecturware.sourceforge.net/site/

Tabelle 3.4: MDA-Werkzeuge

Für die MDA-Werkzeugevaluierung in Kapitel 4 werden ArcStyler und Objecteering/UML als kommerzielle Vertreter untersucht. Da auch einige Open Source Produkte im Bereich MDA-Werkzeuge angeboten werden, wird als bekanntestes Beispiel das AndroMDA Framework näher besprochen.

3.3.4 Codegeneratoren

Einige UML- und MDA-Werkzeuge verfügen über Modell-zu-Code Transformationen, wobei diese nicht veränderbare und auch nicht erweiterbare Transformationen darstellen. Der generierte Code beschränkt sich auf Schnittstellendefinitionen und Klassendefinitionen. Die Codegenerierung wird auch nur für eine beschränkte Menge von Plattformen (meist Java und C#) angeboten. Da die Transformationen nicht verändert werden können, sind alle anderen Plattformen für die Implementierung des Systems ausgeschlossen. Die aktuellen UML-Werkzeuge unterstützen in ihren Professional Editions die Codegenerierung für Java und C# aus UML-Klassendiagrammen. Der generierte Code besteht meistens nur aus Klassen-, Attributen- und Methodendefinitionen.

Ein weiterer Bereich von Codegeneratoren entsteht gerade für das Eclipse Framework. Sie basieren vorwiegend auf zwei Eclipse Tool Projects, nämlich *Eclipse Modeling Framework*[1] (*EMF*) und *Graphical Editing Framework*[2] (*GEF*). EMF verfügt über ein eigenes Metamodell namens Ecore, das große Ähnlichkeiten zu MOF aufweist. Weiters besitzt EMF ein eigenes Codegenerator-Framework namens *Java Emitter Templates* (*JET*). Mittels GEF können mit relativ geringem Aufwand ausgereifte grafische Modellierungseditoren entwickelt werden. Ein weiteres Eclipse Tool Project namens *UML2 Project*[3] bietet eine Implementierung des UML2 Metamodells

---

[1] kostenlos beziehbar über http://www.eclipse.org/emf (Stand: 1.1.2005)

[2] kostenlos beziehbar über http://www.eclipse.org/gef (Stand: 1.1.2005)

[3] kostenlos beziehbar über http://www.eclipse.org/uml2 (Stand: 1.1.2005)

basierend auf EMF. Leider wird durch das UML2 Project nur das Metamodell realisiert, ein grafischer Editor jedoch nicht.
Zur Zeit werden Codegeneratoren basierend auf diesen Frameworks angeboten, die aus UML Klassendiagrammen einige Klassendefinitionen ableiten, aber keine MDA-Konzepte umsetzen. Jedoch besitzen diese Frameworks und die Eclipse-Plattform ein enormes Potential für die Entwicklung von MDA-Werkzeugen durch Plugins.

In dieser Arbeit werden Werkzeuge der Kategorie Codegeneratoren nicht näher untersucht, da diese keine MDA-Konzepte implementieren und nur begrenzt zur Codegenerierung bzw. zur modellgetriebenen Softwareentwicklung eingesetzt werden können. Deshalb wird auch kein Werkzeug dieser Kategorie im Kapitel 4 evaluiert.

Kapitel 4

Evaluierung

Die wichtigsten Vertreter der im Kapitel 3.3 definierten Werkzeugkategorien werden in diesem Kapitel genauer beschrieben. Für den Executable UML Werkzeugbereich werden Nucleus BridgePoint Development Suite (Unterkapitel 4.1) und iUML/iCCG (Unterkapitel 4.2) ausgewählt. Im Bereich technologiespezifische Werkzeuge wird OptimalJ (Unterkapitel 4.3) detailliert besprochen. Als Vertreter der eigentlichen MDA-Werkzeuge werden ArcStyler (Unterkapitel 4.5) und Objecteering/UML (Unterkapitel 4.6) als kommerzielle Produkte, AndroMDA (Unterkapitel 4.4) als Open Source-Produkt ausgewählt.
Für jedes Werkzeug wird auf die wichtigsten Funktionen, wie Modellerstellung, Transformationen und ihre technischen Realisierungen im Werkzeug, näher eingegangen und danach eine Evaluierung des Tools anhand der in Kapitel 3.2 aufgestellten Kriterien vorgenommen.

Abschließend wird im Unterkapitel 4.7 eine übersichtliche Bewertung der besprochenen Tools anhand des aufgestellten Kriterienkataloges in Tabellenform durchgeführt. Die tabellarische Bewertung soll einen Vergleich der aktuellen MDA-Werkzeuge auf einen Blick ermöglichen.

4.1 Nucleus BridgePoint

Hersteller	Accelerated Technology, Embedded Systems Division of Mentor Graphics Corporation
Webseite	http://www.acceleratedtechnology.com
Version	Nucleus BridgePoint Development Suite 6.1, Windows Edition

Nucleus BridgePoint Development Suite (im Weiteren kurz mit Nucleus BridgePoint bezeichnet) wird von der Embedded Systems Abteilung (genannt Accelerated Technology) der Firma Mentor Graphics Corporation entwickelt. Die Umgebung findet seine Anwendung in den Gebieten Echtzeitsysteme, Embedded Systems und Simulationssoftware. Nucleus BridgePoint wurde bereits für die Entwicklung von zahlreichen Systemen in den Bereichen Flugzeugsteuerung, medizinische Systeme und Telekommunikation erfolgreich eingesetzt. Es werden Versionen für Windows 98/2000/ME/XP und Solaris angeboten. Für die Evaluierung wird die Version 6.1 Windows Edition eingesetzt. Diese Version ist kostenlos über die Homepage der Accelerated Technology Abteilung beziehbar und eine für ein Monat gültige Seriennummer wird von einem Betreuer der Firma Mentor Graphics per e-mail zugesandt. Leider sind in dieser Version keine *Model Compiler* enthalten. Diese sind gesondert von der Firma Mentor Graphics zu beziehen und werden nicht kostenlos zur Verfügung gestellt. Daher wird in dieser Arbeit auf Model Compiler nur beschränkt eingegangen.

In Nucleus BridgePoint wird die Entwicklung von Softwaresystemen in vier Phasen eingeteilt. In der ersten Phase wird ein PIM mit Hilfe der Executable UML Notation erstellt. Phase Zwei beinhaltet die Verifikation und den Test des in Phase Eins erstellten Modells auf Modellebene, d.h. das Modell braucht dazu nicht in Code transformiert werden. Die dritte Phase stellt die Transformation des Modells in eine konkrete Implementierung, wie C oder C++, dar. In der vierten Phase ist es möglich, das transformierte Modell zu debuggen, jedoch nicht auf den Elementen der Codeebene basierend, sondern auf den Elementen der Modellebene. Ist die EntwicklerIn mit der Qualität und der Funktionalität des erstellten Systems zufrieden, werden die automatisch generierten Quelltexte kompiliert und in eine ausführbare Datei, wie in einem herkömmlichen Softwareentwicklungsprozess, transformiert.

Jede Phase des Entwicklungsprozesses wird durch ein entsprechendes Werkzeug der Nucleus BridgePoint-Umgebung unterstützt. Ein Model Repository ermöglicht die nahtlose Zusammenarbeit der verschiedenen Tools und dadurch eine integrierte modellbasierte Softwareentwicklungsumgebung. Abbildung 4.1 zeigt den Aufbau der Nucleus BridgePoint-Umgebung. Dabei sei noch einmal darauf hingewiesen, dass die Model Compiler nicht standardmäßig in der Nucleus BridgePoint-Umgebung enthalten sind und zusätzlich bezogen werden müssen, was einen erhöhten finanziellen Aufwand bedeutet.

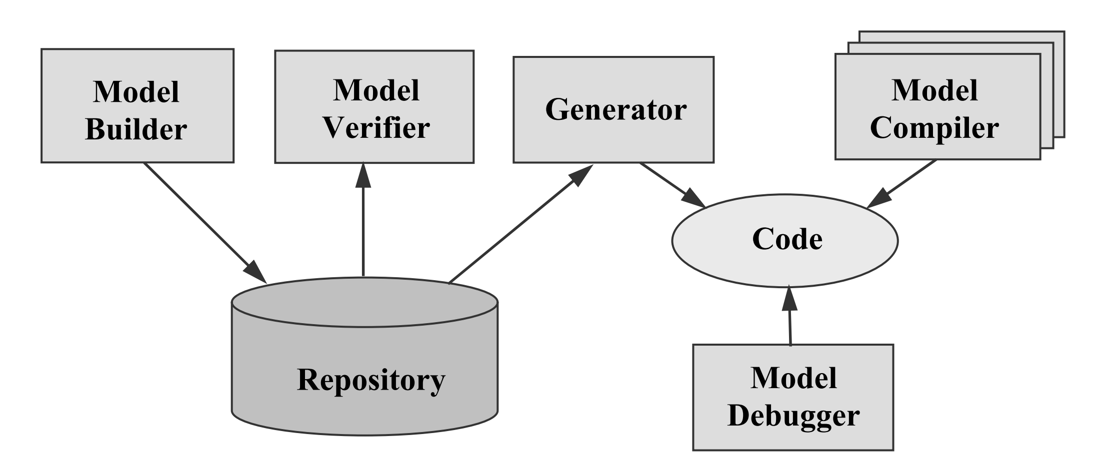


Abbildung 4.1: Aufbau der Nucleus BridgePoint Umgebung nach [AT04a]

Der *Model Builder* ist für die erste Phase zuständig, denn mit seiner Hilfe ist die Erstellung von Executable UML Modellen möglich. Der *Model Verifier* ist für die zweite Phase einzusetzen, denn er unterstützt die Verifikation von Modellen. Der *Generator* nutzt die Transformationspezifikationen der Model Compiler (diese Beschreiben die Modell-zu-Code Transformationsregeln) und die im Model Repository gespeicherten Modelle, um aus diesen zwei Eingaben den gewünschten Code zu generieren. Somit sind für die dritte Phase der Generator und die Model Compiler die unterstützenden Komponenten. Für die vierte Phase wird der *Model Debugger* eingesetzt, um das übersetzte Modell, möglicherweise kombiniert mit Legacy Code, zu testen.

Im Model Builder (Abbildung 4.2) können Modelle mit Hilfe von Domänen-Paketdiagrammen, Klassendiagrammen, Zustandsdiagrammen und Aktionsbeschreibungen erstellt werden. Aus diesen primären, durch die BenutzerIn erstellbaren, Diagrammen können weitere Diagrammarten durch das Werkzeug automatisch erstellt werden, wie z.B. Objekt-Kollaborationsdiagramme oder Zustands-Transitions-Tabellen. Diese sekundären Diagrammarten sollen der BenutzerIn einen besseren Überblick bieten und verschiedene Sichten auf das modellierte System gewähren.

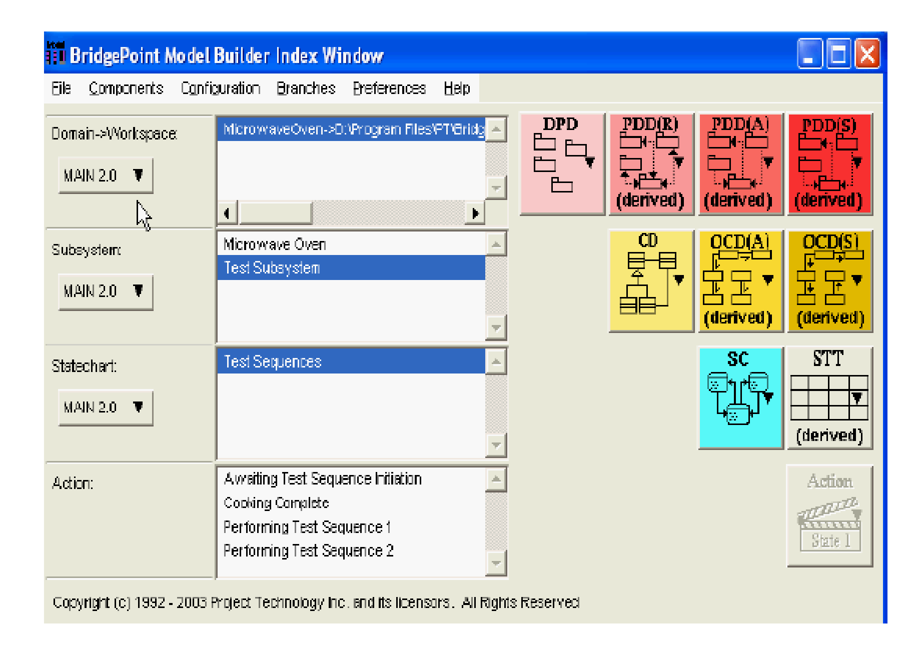


Abbildung 4.2: Hauptansicht des Model Builder

Der Model Builder überprüft die Modelle auf die korrekte Verwendung der Modellelemente und die Syntax der in *Object Action Language* (*OAL*) [AT04b] spezifizierten Aktionsbeschreibungen. OAL ist eine Sprache, welche die Action Semantic für UML 1.5 umsetzt. Zusätzlich zu den Modellierungsaspekten bietet der Model Builder Funktionen für das Konfigurations- und Versionsmanagement und unterstützt die automatische Generierung von Dokumentationen.

Sobald ein Modell im Model Builder erstellt und auf syntaktische Korrektheit überprüft wurde, kann dieses Modell im Model Verifier verwendet werden. Der Model Verifier wird für die interaktive Ausführung und Verifikation von durch den Model Builder erstellten Modelle eingesetzt. Die Ausführung auf Modellebene ermöglicht eine frühe Fehlererkennung im Softwareentwicklungsprozess. Der Model Verifier erlaubt den EntwicklerInnen, die Definition und Ausführung von einfachen Unit-Tests bis hin zu umfangreichen Testsuiten. Abbildung 4.3 zeigt das Hauptfenster des Model Verifiers. Zu Beginn muss ein erstelltes Modell ausgewählt werden. Danach muss der Anfangszustand des Systems erstellt werden, d.h. es müssen die Initialwerte - Instanzen von Klassen und Ereignissen - erzeugt werden. Sind die Initialwerte für die Simulation definiert, kann das System auf drei unterschiedlichen Ausführungsgeschwindigkeiten ausgeführt werden. Durch *Run* wird das System in einem Schritt durchlaufen, ohne auf Benutzereingaben oder Zeitereignisse zu warten. Mittels *Jog* wird pro Zeiteinheit – diese Zeiteinheit kann von der BenutzerIn frei definiert werden – nur ein Ereignis ausgelöst. Mit der Einstellung *Step* steuert die BenutzerIn die Ausführungsgeschwindigkeit. Es wird immer nur ein Ereignis ausgelöst,

danach wird solange gewartet, bis die BenutzerIn durch das Aktivieren der *Tick*-Schaltfläche den nächsten Schritt startet. Die Durchführung einer Simulation wird in einer Textdatei mit Endung .sim_log mitprotokolliert. In diesen Dateien werden die Ereignisreihenfolgen, das Erzeugen bzw. Löschen von Objekten und die Zustände der Objekte festgehalten.

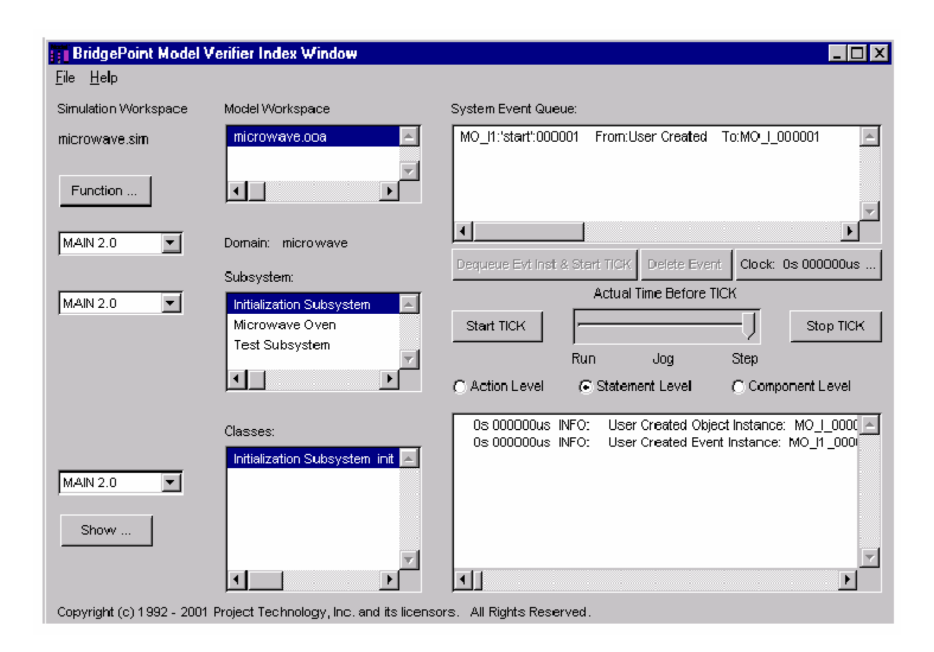


Abbildung 4.3: Hauptansicht des Model Verifiers

4.2 iUML/iCCG

Hersteller	Kennedy Carter
Webseite	http://kc.com
Version	iUMLite 2.20 für Windows

Die Firma Kennedy Carter vertreibt die beiden Werkzeuge iUML und iCCG. iUML steht für Intelligent UML und ist für die Erstellung und Simulation von Executable UML-Modellen verantwortlich. iCCG steht für Intelligent Configurable Code Generator und ist für die Transformation von Executable UML-Modellen zu Code zuständig. Es sind Versionen von iUML/iCCG für Windows NT4/2000/XP und für Solaris ab Version 2.5 verfügbar. Die Anwendung von iUML/iCCG befindet sich, wie bei Nucleus BridgePoint, im Embedded Systems-Bereich, in dem bereits mehrere erfolgreiche Projekte umgesetzt wurden.

Für die Evaluierung wird eine Testversion von iUML, genannt iUMLite, eingesetzt. Diese Testversion ist in einigen Bereichen nur eingeschränkt verwendbar. Die für die Evaluierung relevanteste Einschränkung ist, dass Executable UML-Modelle nur durch den iUML-Simulator ausgeführt werden können, da in der Evaluationsversion keine Codegenerierungsfunktion vorgesehen ist. Deshalb wird auch das iCCG-Werkzeug für die Konfiguration des Codegenerators nicht mitgeliefert. Da keine kostenlose Version des iCCG-Tools angeboten wird, wird in dieser Arbeit die Codegenerierung in iUML/iCCG nur theoretisch, basierend auf den Dokumentationen zu iUML [KC02a] und iCCG [KC02b], betrachtet.

4.2.1 iUML

Die Evaluationsversion iUMLite besteht aus zwei Hauptkomponenten, nämlich aus dem *iUML Modeler* und dem *iUML Simulator*. Der iUML Modeler bietet der AnwenderIn die Möglichkeit, konsistente Executable UML Modelle zu erstellen und in einer Modelldatenbank zu speichern. Der iUML Simulator wird eingesetzt, um die mit dem iUML Modeler erstellten Modelle auszuführen. Dafür werden die Modelle durch das Werkzeug in C Code transformiert und sind somit im iUML Modeler ausführbar und testbar. Der für den Einsatz von iUML vorgeschlagene Entwicklungsprozess ist äquivalent zu dem von Nucleus BridgePoint definierten Entwicklungsprozess. Der einzige Unterschied besteht darin, dass iUML keinen Model Debugger anbietet und somit die letzte Phase des Entwicklungsprozesses wegfällt. iUML und iCCG basieren auf der *Action Specific Language* (*ASL*). ASL ist eine implementierungsunabhängige Sprache für die Spezifikation von Aktionen in Executable UML-Modellen. Weiters ist ASL kompatibel zu Action Semantics für UML. ASL wird in iUML nicht nur für die Spezifikation von Aktionen eingesetzt, sondern auch für die Definition von Einschränkungen. In iCCG wird ASL für die Definition von Transformationsspezifikationen eingesetzt. Für nähere Informationen zu ASL wird auf [KC01] verwiesen.

iUML bietet umfangreiche Testmöglichkeiten von Systemen auf Modellebene. Um ein auf Executable UML basierendes Modell testen zu können, müssen vorher die Anfangsbedingungen des Modells, wie Objekte, Objektzustände und Signale, bestimmt werden. Zusätzlich müssen Testmethoden vorhanden sein, die beschreiben, wie ein Modell simuliert werden soll (wie z.B. durch die Angabe der Sendesequenz von Signalen). Im iUML Modeler werden die Anfangsbedingungen und die Testmethoden in ASL formuliert.

Der iUML Simulator nutzt die durch den iUML Modeler erstellten Modelle (inklusive der Anfangsbedingungen und der Testmethoden), um die Modelle durch ihre Ausführung zu verifizieren. Der BenutzerIn wird durch den iUML Simulator eine grafische Benutzeroberfläche geboten, welche die Simulationen zu kontrollieren erlaubt und zusätzlich Objekte, Objektzustände, Signale und ausgeführten ASL-Code visualisiert. Die Benutzeroberfläche des iUML Simulators (Abbildung 4.4) besteht aus einem Hauptfenster für die Steuerung der Modellausführung (linker Bereich der Abbildung 4.4) und aus einblendbaren Tabellen, die Informationen über Objekte, Signale und ASL-Variablen beinhalten (rechter Bereich der Abbildung 4.4). Im Hauptfenster kann zwischen automatischer Ausführung (d.h. bis keine Signale mehr vorliegen) oder benutzerdefinierter Ausführung (d.h. die BenutzerIn kann wählen, ob bis zum nächsten Signal, bis zur nächsten Operationsausführung oder bis zur nächsten Zeile ASL-Code in einem Schritt fortgefahren wird) gewählt werden. Falls gewünscht, kann die Modellausführung in einer Log-Datei mitprotokolliert oder aufgezeichnet werden. Aufgezeichnete Modellausführungen können im iUML Simulator abgespielt werden und bieten dadurch eine hervorragende Dokumentationsmöglichkeit für Tests.

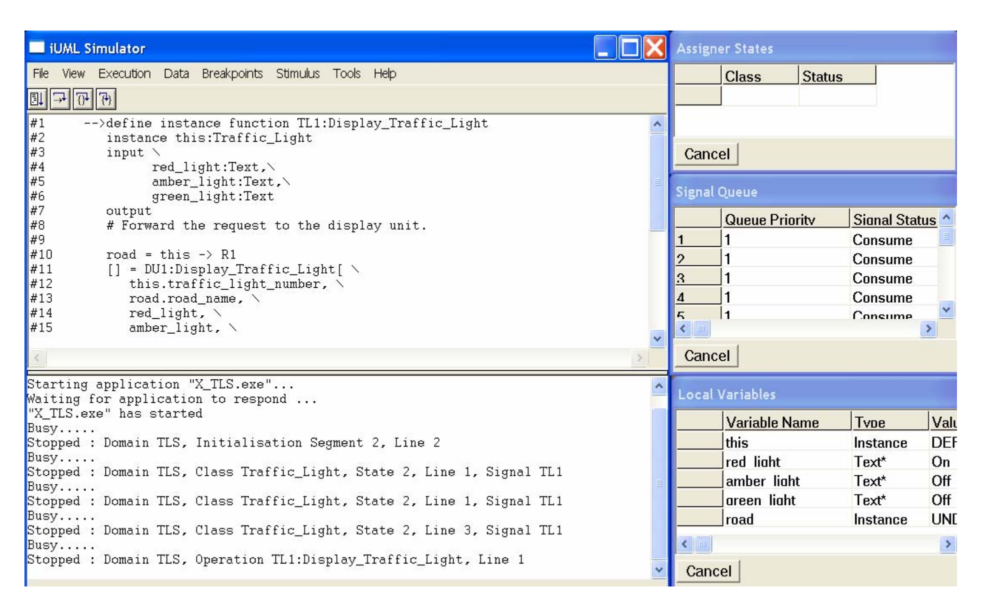


Abbildung 4.4: Benutzeroberfläche des iUML-Simulators

4.2.2 iCCG

iCCG stellt ein Codegenerierungsframework dar, das den EntwicklerInnen die Entwicklung von eigenen Transformationen (Executable UML-Modelle zu Code) oder die Anpassung von vorhanden Transformationen erlaubt. Die Transformationen werden in ASL verfasst und basieren auf dem Metamodell von Executable UML. In

$FORMAT-Blöcke (siehe Abbildung 4.5) ist es möglich Skeleton-Code für die zu generierenden Dateien in ASL zu definieren. iCCG verfügt über einen eigenen Codegenerator, der die in ASL geschriebenen Transformationsregeln aufnimmt und daraus einen Transformationsgenerator für das eigene Projekt generiert. Dieser iCCG-Codegenerator stellt ein zu CompilerCompiler ähnliches Konzept dar. Der Codegenerator für das eigene Projekt ist eine ausführbare Datei, die im iUML Modeler erstellte Executable UML- Modelle in Code transformiert. Abbildung 4.5 verdeutlicht das eben erklärte Konzept des iCCG-Codegenerierungsframeworks.

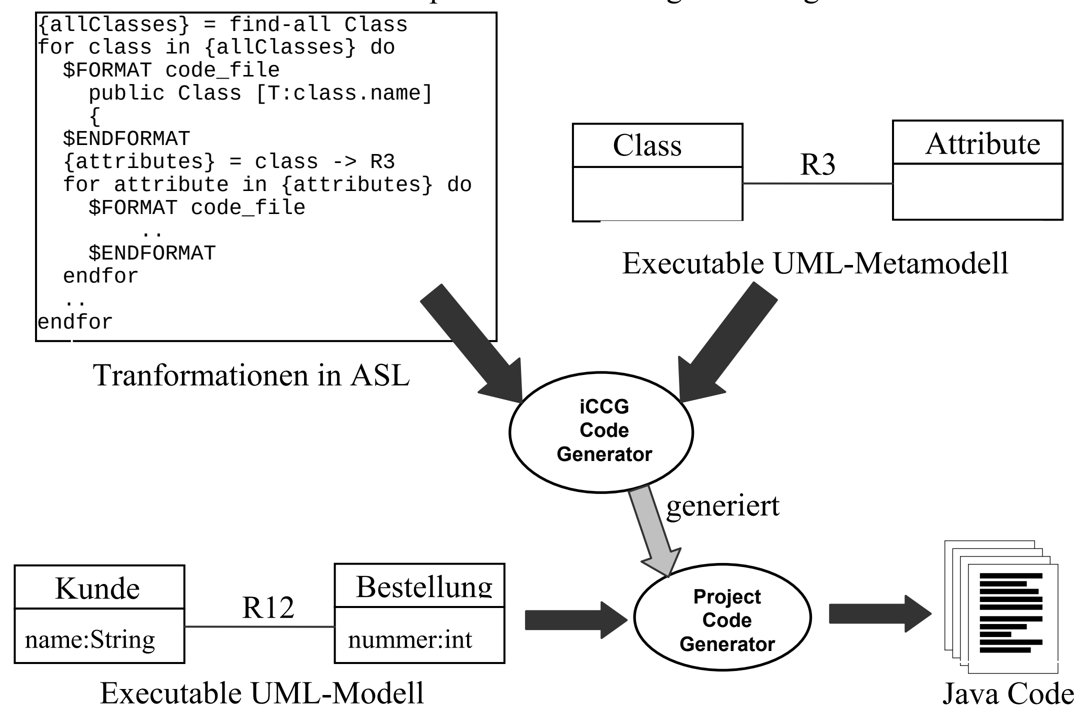


Abbildung 4.5: iCCG-Codegenerierungsframework basierend auf [KC04]

4.3 OptimalJ

Hersteller	Compuware Corporation
Webseite	http://www.compuware.com
Testversion	OptimalJ 3.1 Professional Edition, Windows Version

4.3.1 Einführung

OptimalJ ist eine Enterprise Application-Entwicklungsumgebung, in der einsatzfähige J2EE-Applikationen direkt aus UML-Modellen erzeugt werden. MDA-Konzepte werden mit ausgereiften Patterns [CC04a] für die Erstellung und Transformation von

Modellen kombiniert. Vertrieben wird das Tool von der Firma Compuware Corporation und befindet sich zurzeit in der Entwicklungsversion 3.1. OptimalJ ist in drei verschiedenen Editionen (Architecture Edition, Professional Edition und Developer Edition) erhältlich, wobei jede Version auf eine bestimmte Rolle der im Softwareentwicklungsprozess involvierten Mitarbeiter fokussiert.

Die Developer Edition wird für EntwicklerInnen angeboten, die Arbeiten an J2EE-Artefakten vornehmen, jedoch ohne auf Modellierungsfeatures zurückzugreifen. Dabei soll die EntwicklerIn nicht mit Infrastrukturcode konfrontiert werden, sondern die Geschäftslogik implementieren und dadurch die Anwendung vervollständigen. Die Professional Edition erweitert die Developer Edition und ist für erfahrene EntwicklerInnen und Software-DesignerInnen vorgesehen, die die Analyse und das Design anhand von Modellen durchführen. Die erstellten Modelle werden zuerst in auf J2EE spezialisierte Modelle und danach in J2EE-Artefakte transformiert. Die Architecture Edition erweitert die Professional Edition und ist für den Gebrauch von auf die J2EE-Plattform spezialisierten ArchitektInnen vorgesehen, die neue Modelltypen oder neue Patterns erstellen bzw. mitgelieferte Modelltypen oder Patterns modifizieren.

Unabhängig von der Edition läuft OptimalJ auf Windows NT/2000/XP, Sun Solaris und Linux. Prinzipiell sollte OptimalJ jedoch auf jeder mit einer Java Virtual Machine ausgestatteten Plattform laufen. Auf der Applikationsserverebene sollte OptimalJ mit jeder J2EE kompatiblen Plattform zusammenarbeiten. Mit OptimalJ werden folgende Open Source Tools mitgeliefert: Tomcat als Web-, JBoss als Applikations- und Solid als Datenbankserver. Diese mitgelieferten Technologien werden verwendet, um eine in OptimalJ integrierte Testumgebung zu schaffen. Ab Version 3.1 wurde die Testumgebung für weitere Applikationsserver, wie z.B. BEA's Weblogic 8.1 oder IBM's WebSphere 5, erweitert. Über konfigurierbare Systemeigenschaften lassen sich beliebige Plattformen für die Testumgebung festlegen. Diese Konfigurationsmöglichkeit unterstützt die automatische Erstellung von Deployment Descriptors für spezifische Plattformen.

4.3.2 Abstraktionsebenen in OptimalJ

Wie durch das MDA-Paradigma gefordert, unterscheidet OptimalJ bei der Entwicklung eines Softwaresystems zwischen Modellen und der Implementierung. Um diese Trennung entsprechend umzusetzen, wird in OptimalJ eine Applikation in drei unterschiedlichen Abstraktionsebenen erstellt. Dabei werden die Ebenen äquivalent zum

MDA-Paradigma eingeteilt, aber OptimalJ bezeichnet die Ebenen mit eigenen Namen. Im Folgenden werden die drei Abstraktionsebenen von Modellen in OptimalJ aufgelistet und beschrieben.

- *Domain-Modell:* Auf dieser Ebene werden fachliche Modelle erstellt, die die Struktur und das Verhalten der zu entwickelnden Systeme plattformunabhängig, d.h. ohne auf Technologiedetails einzugehen, beschreiben. Das Domain-Modell entspricht dem PIM des MDA Paradigmas. Die Domain-Modellebene wird in zwei Unterkategorien (*Domain-Class-Modell* und *Domain-Service-Modell*) eingeteilt. Das Domain-Class-Modell beschreibt durch den Einsatz von Klassendiagrammen die statische Struktur der Applikation. Das Domain-Service-Modell beschreibt die Verhaltensaspekte der Anwendung. Durch das Domain-Service-Modell wird aber nur das Verhalten für Create/Update/Delete der Entitäten abgedeckt.

- *Applikations-Modell:* Diese Ebene definiert die Applikationsarchitektur, basierend auf der J2EE-Plattform, aber ohne auf Details der Programmierung einzugehen. Das Applikations-Modell liegt zwischen dem abstrakten Domain-Modell und dem konkreten Code-Modell. Das Applikations-Modell beschreibt, welche Elemente generiert werden müssen, um eine einsatzfähige J2EE-Applikation zu implementieren. Ein aus dem Domain-Modell generiertes Applikations-Modell besteht aus folgenden drei Modelltypen:

    - *Präsentations-Modell:* Dieses Modell beinhaltet die für die Erstellung einer Web-Benutzerschnittstelle notwendigen Informationen. Durch Wizards können EntwicklerInnen das Präsentations-Modell durch deklarative Ausdrücke verfeinern. Zusätzlich können Eingabedaten durch reguläre Ausdrücke auf bestimmte Formate validiert werden. OptimalJ verwendet das Präsentations-Modell, um die Kommunikation mit der BenutzerIn durch JSPs und Servlets zu realisieren.

    - *Geschäftslogik-Modell:* Das Geschäftslogik-Modell basiert auf der Enterprise Java Beans-Technologie. Dieses Modell kann als Abstraktion für Enterprise Java Beans-Programmcode gesehen werden. OptimalJ verwendet das Geschäftslogik-Modell, um das EJB-Modell mit seinen Elementen wie z.B. Entity Beans oder Session Beans zu erzeugen.

  - *Datenbank-Modell:* J2EE-Architekturen benötigen Technologien, um persistente Informationen zu archivieren. Dafür werden auf Datenbank-Modellebene alle benötigten Datenbankdefinitionen (wie z.B. Tabellen, Spalten oder Primärschlüssel) mittels ER-Diagrammen spezifiziert. Das Datenbank-Modell wird in OptimalJ für die Erzeugung von SQL-Skripten, die für das Erzeugen bzw. Löschen von Tabellen und das Befüllen der Tabellen mit Testdaten verantwortlich sind, verwendet.

- *Code-Modell:* Diese Ebene definiert die Anwendung durch implementierten Code, der direkt aus dem Applikationsmodell erzeugt wird. Dabei werden nicht nur Java-Klassen generiert, sondern auch alle für den Applikations-/Datenbankserver benötigten XML-/SQL-Dateien. Die automatisch generierte Applikation ist sofort ohne Modifikationen in der OptimalJ-Umgebung ausführbar und dadurch testbar. Der Code wird auf Basis von *Design Patterns* nach [Gamm96] erstellt, um den Code leicht verständlich und gut wartbar zu halten. Die beiden letztgenannten Eigenschaften sind äußerst wichtig, da im Code Funktionen von den EntwicklerInnen implementiert werden müssen, die im Domänen- bzw. Applikations-Modell nicht spezifiziert werden können. Um diese manuellen Änderungen am Code vornehmen zu können, ist der automatisch erzeugte Code in frei editierbare und für den Generator geschützte Bereiche (nicht zu verwechseln mit den für ProgrammiererInnen geschützten Bereichen im Kriterienkatalog) unterteilt. Für den Generator geschützte Bereiche sind im Netbeans-basierten internen Editor blau hinterlegt (Abbildung 4.6) und können durch die BenutzerIn nicht verändert werden. Hingegen können frei editierbare Bereiche von der BenutzerIn modifiziert werden und sind im internen Editor weiß hinterlegt (Abbildung 4.6). Zusätzlich zum internen Editor wird für den Code eine Exportfunktion in Projektdateien für die bekanntesten Programmier-IDEs wie Borland JBuilder oder SUN ONE Studio angeboten. Für den Code der Präsentationsschicht wird eine Exportfunktion für Macromedia Dreamweaver angeboten, um der BenutzerIn die Erstellung der Benutzerschnittstelle zu erleichtern.

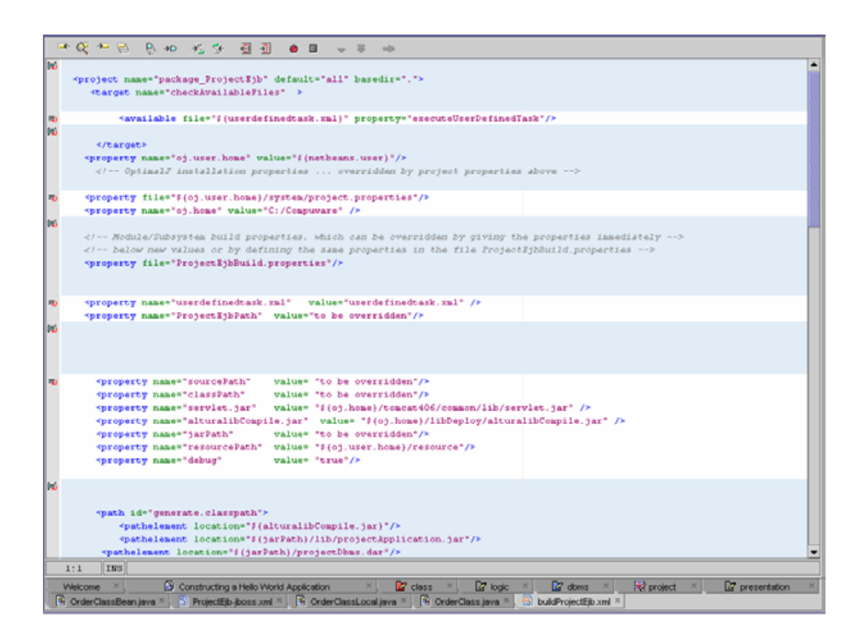


Abbildung 4.6: freie und geschützte Codebereiche in OptimalJ

Da nun die Abstraktionsebenen in OptimalJ ausreichend besprochen wurden, können die Transformationen zwischen den einzelnen Ebenen (Domänen- zu Applikationsmodell und Applikations- zu Codemodell) erläutert werden. Da OptimalJ das MDA-Paradigma mit dem Einsatz von Patterns (Abbildung 4.7) kombiniert, werden daher die notwendigen Transformationen mit einem Pattern-getriebenen Generierungsprozess durchgeführt. In OptimalJ werden aber nicht nur für die Transformationen Patterns eingesetzt, sondern auch für die Struktur der Modelle. Generell kann die Vielzahl an verwendeten Patterns in zwei Hauptkategorien, nämlich in *Transformation-* und *Functional-Patterns*, eingeteilt werden.

- Transformation-Patterns werden für Transformationen zwischen Modellebenen verwendet. Mit ihrer Hilfe werden die Zusammenhänge zwischen Elementen auf Domain- und Applikations-Modell bzw. Applikations- und Code-Modell beschrieben. Transformation-Patterns sind eng mit den definierten Metamodellen von OptimalJ verbunden, da diese die Typen der möglichen Elemente von Modellen beschreiben. Transformation-Patterns werden in *Technology-Patterns* und *Implementation-Patterns* unterteilt.

    - Technology-Patterns werden verwendet, um Modelle in weitere Modelle zu transformieren. Konkrete Anwendung finden sie bei der Transformation von Domain- zu Applikations-Modellen. Technology-Patterns werden in OptimalJ durch sogenannte *Incremental Copiers* implementiert. Incremental Copiers basieren auf der Java-Plattform und besitzen ein deklaratives, regelbasiertes Format. Der

Ablauf einer Transformation mittels Incremental Copiers kann in zwei Phasen eingeteilt werden. Zuerst wird die Struktur des Zielmodells erstellt, danach werden die Werte, basierend auf dem Quellmodell, den bereits erzeugten Elementen des Zielmodells hinzugefügt. In dieser Diplomarbeit soll aber nicht näher auf dieses Konzept eingegangen werden. Für weitere Informationen und Beispiele wird der Leser auf Kapitel 1.4.4 in [CC04c] verwiesen.

- Implementation-Patterns werden verwendet, um Modelle in Code zu transformieren. Angewendet werden sie bei der Transformation von Applikations- zu Code-Modellen. Für die Erstellung von Implementation-Patterns wird eine proprietäre Skriptsprache (Unterkapitel 4.3.3) in OptimalJ geboten.

- Functional-Patterns werden auf einer Ebene genutzt. Für die Domain-Modellebene werden *Domain-Patterns*, für die Applikations-Modellebene werden *Applikations-Patterns* und für die Code-Modellebene werden *Code-Patterns* verwendet.
  Ein Domain-Pattern ist ein UML-Klassendiagramm, das jederzeit wiederverwendet werden kann, wie z.B. ein Klassendiagramm mit den Klassen Kunde, Artikel und Warenkorb für die Spezifikation eines Webshops. Ein neues Domain-Modell kann anhand von Domain-Patterns aus Domain-Pattern-Bibliotheken konstruiert werden. Domain-Patterns sind immer applikations- und implementierungsunabhängig.
  Applikations-Patterns folgen dem selben Prinzip wie Domain-Patterns, jedoch mit dem Unterschied, dass sie auf Applikations-Modellebene angewendet werden und dadurch applikationsabhängig sind.
  Code-Patterns werden auf Code-Modellebene verwendet, um eine hohe Codequalität zu erreichen. Es werden zum Beispiel die gesamten Entwurfsmuster der Gang of Four [Gamm96] bei der Generierung von Java-Code implementiert.

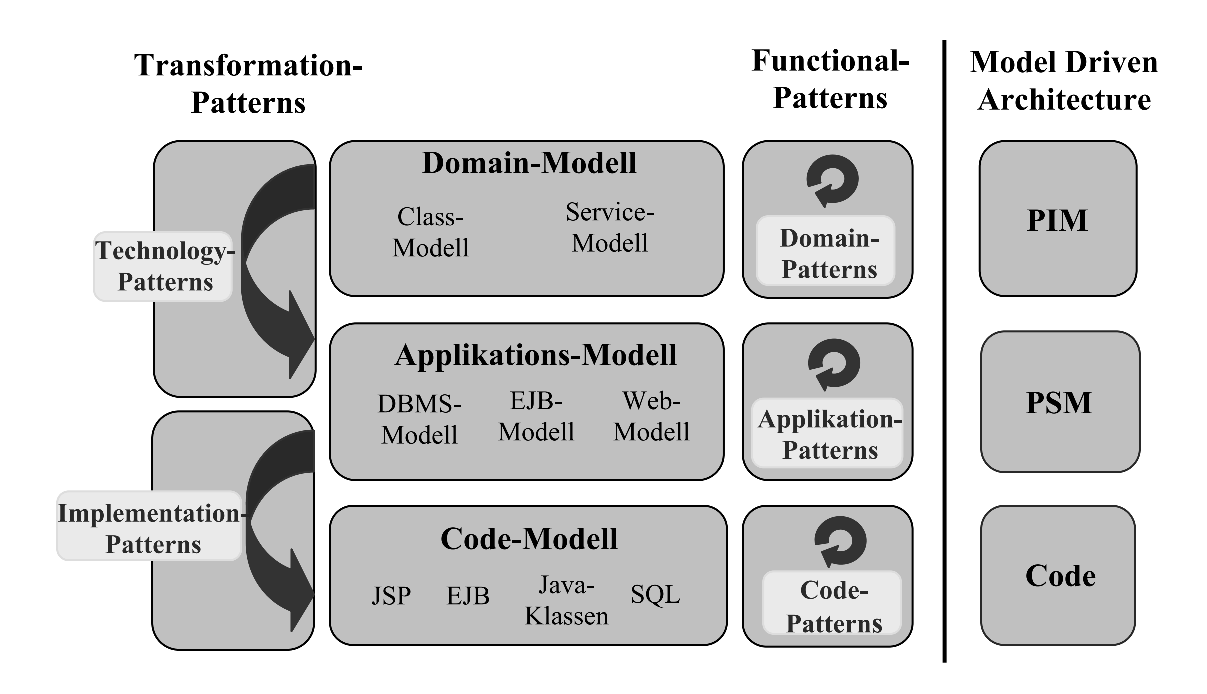


Abbildung 4.7: Abstraktionsebenen und Pattern-Kategorien in OptimalJ nach [CC04]

4.3.3 Anpassungsmöglichkeiten von OptimalJ

Compuware erlaubt den AnwendernInnen die Anpassung von OptimalJ durch ihre OptimalJ Architecture Edition. Die Architecture Edition ermöglicht die Erstellung und Modifikation von Transformation-Patterns und Metamodellen.

In OptimalJ wird jedes Modell, wie z.B. ein konkretes Geschäftslogik- oder Datenbank-Modell, als Instanz des zugehörigen Metamodells, wie z.B. Geschäftslogik- oder Datenbank-Metamodell, werkzeugintern verarbeitet. Für die Definition von Metamodellen wird MOF in der Version 1.4 eingesetzt. Mit Hilfe der Architecture Edition können SoftwarearchitektInnen neue, auf MOF basierende Metamodelle entwerfen oder die von Compuware bereitgestellten Metamodelle für ihre individuellen Anforderungen anpassen. Durch ein erstelltes Metamodell ist die automatische Generierung der zugehörigen Repository API möglich, wobei das MOF-Modell nach Java transformiert wird. Die Repository API wird einerseits verwendet, um Metamodellelemente zu erstellen, zu verändern oder auf ihre Werte zuzugreifen und andererseits, um die auf dem Metamodell basierenden Modelle zu speichern. Die in OptimalJ verwendete Repository API ist mit der Java Metadata Interface Spezifikation zu vergleichen, nur wurde die Repository API speziell für OptimalJ mit zusätzlichen Funktionalitäten ausgestattet. Durch die Repository API kann auf Werte von Modellelementen zugegriffen werden. Dieses Feature wird speziell bei der Erstellung von Transformationen häufig genutzt.

OptimalJ generiert Modelle und Code durch den Einsatz von Transformation-Patterns. Modelle werden in weitere Modelle durch Technology-Patterns transformiert und der Code wird von Modellen durch Implementation-Patterns erzeugt. Mit Hilfe der Architecture Edition können neue Technology- und Implementation-Patterns erstellt und die von Compuware bereitgestellten Transformation-Patterns modifiziert werden. Im Folgenden wird die Erstellung von Implementation-Patterns genauer besprochen.

Der erste Schritt für die Entwicklung eines Implementation-Patterns stellt die Auswahl des in Code zu transformierenden Metamodells (d.h. Quellmetamodell) dar. Sobald die ArchitektIn das relevante Metamodell auswählt, kann mit der eigentlichen Erstellung der Transformationsspezifikation begonnen werden. OptimalJ bietet für den Entwurf von Implementation-Patterns eine proprietäre Sprache, genannt *Template Pattern Language* (*TPL*).

Die Sprache TPL

TPL ist für die Erstellung von Code-Patterns und Transformationen von MOF-basierten Modellen (z.B. Datenbank- oder Präsentations-Modell) zu Syntax-basierten Sprachen (z.B. Java oder XML) konzipiert. In OptimalJ wird ausschließlich die Sprache TPL für die bereitgestellten Transformationen von dem Applikations-Modell zu Java-, XML- und SQL-Artefakten genutzt. TPL-Skripte werden in Java-Klassen kompiliert, die in OptimalJ als Implementation-Pattern-Module importiert werden. Diese Module haben Zugriff auf die Repository API, um die für den Generierungsprozess notwendigen Modellinformationen abzufragen. Die Java-Klassen nehmen auf MOF basierende Modelle als Eingabe und erzeugen Artefakte, wie z.B. XML- oder SQL-Dokumente, als Ausgabe.

Ein TPL-Skript gliedert sich in einen Deklarationsteil und einen Skriptkörper. Der Deklarationsteil spezifiziert den Template-Namen und die Übergabeparameter. Der Skriptkörper wird wiederum in TPL-Anweisungen und Skeleton-Code unterteilt. TPL-Anweisungen dienen der Steuerung des Kontrollflusses (z.B. Schleifen und Bedingungen) und der Kommunikation mit der Repository API. Skeleton-Code-Elemente sind vom Modell unabhängige Textfragmente, die in die Ausgabedatei geschrieben werden. TPL bietet für die Notation von Skeleton-Code und TPL-Anweisungen zwei unterschiedliche Konzepte. Diese beiden Konzepte werden anhand eines sehr einfach gehaltenen Beispiels erklärt. Dabei soll die Transformation

einer UML-Klasse in Java Source Code definiert werden. Es wird aber nur der Name der UML-Klasse berücksichtigt und dem Skript als Parameter übergeben. Auf mögliche Attribute oder Methoden wird nicht eingegangen. Nehmen wir an, die UML-Klasse heißt Termin. Es soll durch die Transformation folgender Java Source Code erstellt werden:

```
public class Termin{

}
```

Im Folgenden werden nun die beiden unterschiedlichen Notationskonzepte anhand des vorher besprochenen Beispiels erklärt werden.

- *Bare TPL:* Mittels Bare TPL werden Skeleton-Code Bereiche als String Literale, wie in Java, gekennzeichnet. Dabei müssen auch die aus Java bekannten Formatierungsregeln für Strings beachtet werden. TPL-Anweisungen werden nicht ausgezeichnet, sondern können ohne besondere Markierung im Skript definiert werden. Bare TPL-Codeteile werden immer in [TPL][/TPL]-Blöcke eingeschlossen.
  Eine mögliche Lösung des vorher besprochenen Beispiels in Bare TPL wird im folgenden Codefragment angegeben:

  ```
  [TPL]
  TEMPLATE PUBLIC GENERATE_JCLASS(String name)
      "public class " name "{"
      "\n"
      "}"
  /TEMPLATE
  [/TPL]
  ```

  Bare TPL sollte verwendet werden, wenn Leerzeichen präzise verwaltet werden sollen. Denn nur Leerzeichen innerhalb von Stringdefinitionen werden in die Ausgabedatei übertragen, hingegen werden Leerzeichen außerhalb von Stringdefinitionen für die Artefaktgenerierung ignoriert. Einen weiteren Anwendungspunkt von Bare TPL stellen Skripte dar, bei denen ein Großteil des Skriptes aus TPL-Anweisungen bestehen und wenig Skeleton-Code vorhanden ist, da keine besondere Auszeichnung von TPL-Anweisungen vorgenommen werden muss.

- *Escaped TPL:* Die Struktur und Syntax von Escaped TPL unterscheiden sich kaum von denen von Bare TPL. Der Unterschied liegt darin, dass TPL-

Anweisungen durch eckige Klammern abgegrenzt werden und Skeleton-Code ohne besondere Kennzeichnung notiert wird.
Eine mögliche Lösung des vorher besprochenen Beispiels in Escaped TPL wird im folgendem Codefragment angegeben:

```
[TEMPLATE PUBLIC GENERATE_JCLASS(Stringname)]
    public class [name]{

    }
[/TEMPLATE]
```

Escaped TPL vereinfacht die Notation von Skeleton-Code, da z.B. keine doppelten Anführungszeichen innerhalb des Skeleton-Codes durch Backslashes entwertet werden müssen. Sobald Skripte zum Großteil aus Skeleton-Code bestehen, ermöglicht Escaped TPL eine einfachere Erstellung und eine bessere Lesbarkeit des Skriptes.

OptimalJ ist mit Editoren für TPL und Java ausgestattet, die Syntax-Highlighting, Codevervollständigung und viele weitere Features unterstützen. Der TPL-Editor visualisiert die vom TPL Compiler gemeldeten Fehler, um der AchritektIn die Fehlersuche zu erleichtern. Wird ein TPL-Skript erfolgreich kompiliert, so kann die erzeugte Java-Klasse im Java-Editor angesehen und sogar verändert werden. Letztgenanntes sollte aber nur in Ausnahmefällen praktiziert werden, da diese Änderungen im TPL-Skript nicht berücksichtigt werden und bei der nächsten Kompilierung des TPL-Skriptes verloren gehen.

4.4 AndroMDA

Hersteller	Matthias Bohlen und das AndroMDA Team (Open Source Projekt)
Webseite	http://www.andromda.org
Testversion	AndroMDA 3.0M2

4.4.1 Einführung

AndroMDA beschreibt sich selbst als ein Open Source Codegenerierungsframework, welches auf Konzepten und Prinzipien von MDA basiert. Das Tool ist kostenlos über

das Open Source Portal SourceForge.net[1] beziehbar. AndroMDA entwickelte sich aus dem UML2EJB Projekt[2], dessen primäre Aufgabe darin bestand, den hohen Schreibaufwand von Infrastrukturcode bei der Entwicklung von Enterprise Java Beans zu reduzieren. Dies wurde, wie der Name UML2EJB schon verrät, durch die automatische Generierung von EJB-Infrastrukturcode aus UML-Klassendiagrammen erreicht.

Mit der kontinuierlichen Verbreitung der modellgetriebenen Softwareerstellung setzten sich die Entwickler des UML2EJB-Projektes ein neues Ziel, nämlich die Generierung von jeder nur denkbaren Implementierung aus plattformunabhängigen Modellen zu ermöglichen. Diese Neuorientierung führte zu einem neuen Projekt namens AndroMDA.

Da AndroMDA kein autonomes Werkzeug, sondern ein Framework darstellt, stützt sich die Funktionalität von AndroMDA auf das Zusammenspiel mehrerer Open Source-Produkte. Einen groben Überblick über die Architektur von AndroMDA zeigt Abbildung 4.8, wobei zwischen dem eigentlichen Codegenerierungsframework und seiner Peripherie (externes UML-Werkzeug) unterschieden wird.

[1] http://sourceforge.net/project/andromda

[2] http://sourceforge.net/projects/uml2ejb

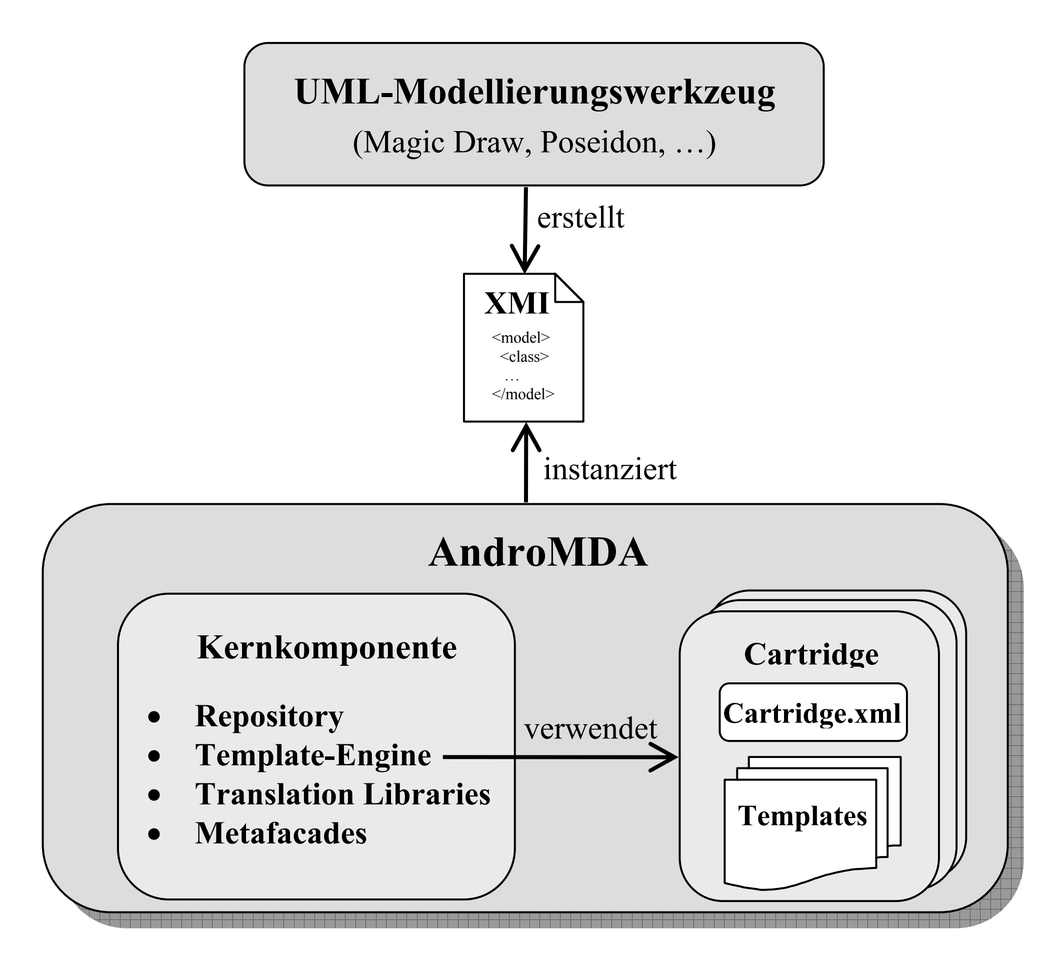


Abbildung 4.8: Aufbau des AndroMDA Frameworks

In der Abbildung 4.8 können drei Hauptkomponenten des AndroMDA Frameworks identifiziert werden, nämlich ein externes Modellierungstool, die AndroMDA- Kernkomponente und die MDA-Cartridges. Wie die Komponenten im Detail aufgebaut sind und welche Aufgaben sie innehaben, wird in den nächsten Abschnitten erläutert.

4.4.2 Modellierungstool

AndroMDA verfügt über kein integriertes Modellierungstool. Daher wird für die Erstellung der Modelle auf Modellierungstools wie No Magic's Magic Draw[1] oder Gentleware's Poseidon[2] verwiesen. In der AndroMDA-Dokumentation [Andr04] wird Magic Draw als Referenzmodellierungswerkzeug für das AndroMDA Framework angegeben und das, obwohl es sich bei Magic Draw um ein kommerzielles Produkt handelt. Gentleware hingegen bietet eine kostenlose Community Edition seines UML-Modellierungswerkzeuges Poseidon zum Download an. Falls gewünscht, können auch weitere UML-Modellierungswerkzeuge verwendet werden,

---

1 http://www.magicdraw.com (Stand: 1.1.2005)

2 http://www.gentleware.com (Stand: .1.1.2005)

nur sollte man darauf achten, dass die UML- und die XMI-Version des UML-Modellierungswerkzeuges von AndroMDA unterstützt werden. Zusätzlich müssen die eingesetzten UML-Modellierungswerkzeuge das Anfügen von Stereotypen und Schlüsselwort/Wert-Paaren unterstützen, da diese Informationen für den Transformationsprozess benötigt werden. Um die Modelle für AndroMDA nutzbar zu machen, müssen diese in eine XMI-Datei exportiert werden. Deshalb ist die letzte Anforderung an die UML-Modellierungswerkzeuge eine Export- bzw. Speicherungsmöglichkeit der Modelle in XMI-Dateien.

4.4.3 Kernkomponente

Die Kernkomponente ist für die Koordination der einzelnen Werkzeuge des AndroMDA-Werkzeuges verantwortlich und wird entweder durch ein ANT Build-Skript[1] oder durch ein MAVEN-Plugin[2] aufgerufen. Speziell für das Einbinden der MDA-Cartridges, das Lesen der XMI Datei, den Aufbau eines abstrakten Syntaxbaumes für die eingelesene XMI-Datei (instanziertes Metamodell) und die Durchführung der Transformationen der Modellelemente greift AndroMDA auf etablierte Open Source-Technologien, soweit diese vorhanden sind, zurück. Die Kernkomponente kontrolliert bzw. ermöglicht das Zusammenspiel der einzelnen Technologien.

Gleichzeitig stellt die Unabhängigkeit von bestimmten Technologien eines der wichtigsten Architekturprinzipien von AndroMDA dar. Über exakt definierte Schnittstellen sind neue Produkte in das AndroMDA Framework integrierbar und somit können die mitgelieferten Produkte einfach substituiert werden. Damit soll die zukünftige Entwicklung von AndroMDA gesichert werden und nicht von der Entwicklung der einzelnen Produkte abhängen.

---

[1] www.ant.org (Stand: 1.1.2005)

[2] www.maven.org (Stand: 1.1.2005)

Open Source-Produkte	Website	Aufgabe im AndroMDA Framework
Netbeans MetaData Repository	http://mdr.netbeans.org	übernimmt das Einlesen der XMI-Datei
Velocity	http://jakarta.apache.org/velocity	fungiert als erster Code-generator
XDoclet	http://xdoclet.sourceforge.net	fungiert als zweiter Code-generator

Tabelle 4.1: mitgelieferte Produkte des AndroMDA Frameworks

Tabelle 4.1 listet die im AndroMDA Framework mitgelieferten Open Source-Produkte und ihre Aufgabenbereiche im AndroMDA Framework auf. Im Folgenden werden die Grundfunktionen dieser Produkte und ihre Aufgaben im AndroMDA Framework genauer beschrieben.

Netbeans MetaData Repository

Im Rahmen des Netbeans-Projektes[1] von Sun Microsystems wird als Teilaufgabe das Netbeans MetaData Repository (MDR) entwickelt. MDR kann entweder als eigenständiges Werkzeug oder in Frameworks verwendet werden. Da MDR den MOF-Standard der OMG umsetzt, ist es möglich, Instanzen jedes MOF-basierenden Metamodells zu laden und zu speichern [Matu03]. Modelle können in MDR importiert oder von MDR exportiert werden. Dies wird durch XML Streams basierend auf dem XMI-Standard ermöglicht. Auf die importierten Modelle kann über das *Java Metadata Interface* (*JMI*, [JCP02]) zugegriffen bzw. können Werte der Modellelemente über diesen Standard modifiziert werden.

AndroMDA nutzt das MDR, um exportierte Modelle in XMI-Form zu verarbeiten und einen abstrakten Syntaxbaum (instanziertes Metamodell) im Speicher aufzubauen, um den Transformationen Zugriff auf die Modellelemente zu gewähren.

Velocity

Velocity ist eine Open Source Template Engine und über Apache Jakarta kostenlos beziehbar. Die Templates werden in *Velocity Template Language* (*VTL*), einer einfach gehaltenen aber durchaus mächtigen Skriptsprache, geschrieben.

[1] http://www.netbeans.org (Stand: 1.1.2005)

Für den Transformationsprozess werden eine Templatedatei und eine Kontextdatei als Eingaben benötigt. Durch die Transformation wird eine Datei produziert, die ein durch die Templatedatei beschriebenes Format besitzt. Während eines Transformationsprozesses werden bestimmte Stellen (Generierungsvariablen) im Template durch Daten aus der Kontextdatei substituiert. Beispielsweise werden Werte von den Modellelementen in die Templates eingesetzt und dadurch wird der gewünschte Code generiert [Bend04].

AndroMDA nutzt Velocity, um aus dem abstrakten Syntaxbaum Quelltext, wie z.B. Java-Klassen oder SQL-Skripte, zu generieren. Da Velocity als erster Codegenerator im AndroMDA Framework verwendet wird, können für den zweiten Codegenerator (XDoclet) Metadaten im Quelltext angegeben werden. Damit ist es möglich, die Transformationsdefinitionen einfacher und übersichtlicher zu gestalten.

XDoclet

XDoclet ist ein Open Source Codegenerator, der über das Open Source Portal SourceForge.net kostenlos bezogen werden kann. Die notwendigen Informationen für den XDoclet Codegenerator werden über sogenannte XDoclet Tags als Kommentarform im Quelltext hinzugefügt. Mithilfe von XDoclet können diese Metadaten ausgewertet und daraus Artefakte, wie z.B. Deployment Descriptors oder *remote*- und *local*-Schnittstellen für Enterprise Java Beans, generiert werden.

Die automatische Erzeugung wird durch Templates gesteuert, die die dafür notwendigen Informationen aus den XDoclet Tags und aus dem Quelltext der Dateien beziehen. Ein Grund für die Entstehung des XDoclet-Projektes war die mühsame Entwicklung von EJB-Komponenten. Da eine Entity Bean bis zu sieben Dateien benötigt und sich eine Änderung in einer Datei auf mehrere Dateien auswirkt, wurde die Forderung nach automatisierter Erstellung und einfacherer Handhabung stärker. XDoclet löst dieses Problem, indem nur mehr eine Datei, die alle relevanten Informationen als Metadaten enthält, erstellt wird und die restlichen Dateien werden daraus automatisch erzeugt.

AndroMDA nutzt XDoclet als zweiten Codegenerator, um die Transformationen für Velocity zu vereinfachen. Die erzeugten Dateien des ersten Generators (Velocity) dürfen XDoclet Tags enthalten. Aus diesen Metadaten erzeugt XDoclet die restlichen Artefakte, die für eine lauffähige Anwendung benötigt werden.

4.4.4 MDA-Cartridges

MDA-Cartridges definieren die Transformationen von Modellelementen, die durch Stereotype und Schlüsselwort/Wert Paare gekennzeichnet wurden, in beliebige Textdateien (meistens Quelltext oder Verteilungsinformationen). Mit der Standardinstallation werden Cartridges für folgende Plattformen bereitgestellt:

Bezeichnung	**Beschreibung**
BPM4Struts	ermöglicht die Geschäftsprozessmodellierung für Jakarta Struts, woraus Webseiten, Form- und Aktionsklassen generiert werden können
EJB	generiert Container Managed Persistent Entity Beans und Session Beans
Hibernate	generiert Code für Objekt/Relational-Mappings durch Hibernate
Java	generiert Java-Quelltextdateien
Spring	generiert Artefakte für das Spring Framework
WebService	generiert Web Service Descriptor Dateien für Apache Axis
XmlSchema	generiert ein XML Schema aus einem UML-Klassendiagramm

Tabelle 4.2: MDA-Cartridges des AndroMDA Frameworks

Für jede MDA-Cartridge gibt es eine vordefinierte Menge von Stereotypen und Schlüsselwort/Wert-Paaren, die für die Steuerung der Codegenerierung unbedingt benötigt werden. In Tabelle 4.3 wird für die *EJB-Cartridge* ein kurzer Auszug aus den vorhandenen Stereotypen angegeben.

Stereotype	**Modellelement**	**Beschreibung**
«Entity»	Classifier	Classifier wird als CMP Entity Bean abgebildet
«Service»	Classifier	Classifier wird als Session Bean abgebildet
«FinderMethod»	Operation	Operation wird als Finder-Methode abgebildet

Tabelle 4.3: Auszug aus dem Stereotypverzeichnis der EJB-Cartridge

4.4.5 Ablauf eines Entwicklungsprozesses mit AndroMDA

Da die interne Struktur des Frameworks ausreichend beschrieben wurde, können nun die Aktivitäten eines Entwicklungsprozesses, der AndroMDA als Entwicklungsframework verwendet, definiert werden. Am Beginn jedes Projektes müssen Cartridges für die einzusetzenden Technologien (Plattformen) identifiziert werden. Sind entweder mit AndroMDA mitgelieferte oder selbsterstellte Cartridges, welche die identifizierten Plattformen unterstützen, vorhanden, kann sofort mit der Erstellung des Sys-

tems begonnen werden. Sollten aber keine Cartridges für die Plattformen vorhanden sein, müssen neue Cartridges erstellt werden. Sobald die benötigten Cartridges vorhanden sind, wird mit der Erstellung der UML-Diagramme in einem externen UML Tool begonnen. Die fertigen Diagramme werden mit Cartridge-konformen Stereotype und Schlüsselwort/Wert-Paaren verfeinert und danach in eine XMI-Datei exportiert bzw. gespeichert. Nun kommt AndroMDA zum Einsatz, welches über ein ANT-Skript oder Maven Plugin aufgerufen wird. AndroMDA liest mit Hilfe von MDR die XMI-Datei und erstellt ein Objektmodell der Diagramme. Velocity greift auf das Objektmodell zu und kann mittels definierter Schablonen einen ersten Codeentwurf, der Metadaten enthält, erzeugen. XDoclet analysiert die von Velocity generierten Ausgabedateien und erzeugt daraus weitere Dateien, wie z.B. Schnittstellenspezifikationen und Deployment Descriptors. Somit ist der gesamte Infrastrukturcode erzeugt und es müssen nur mehr die Geschäftslogikmethoden manuell ausprogrammiert werden.

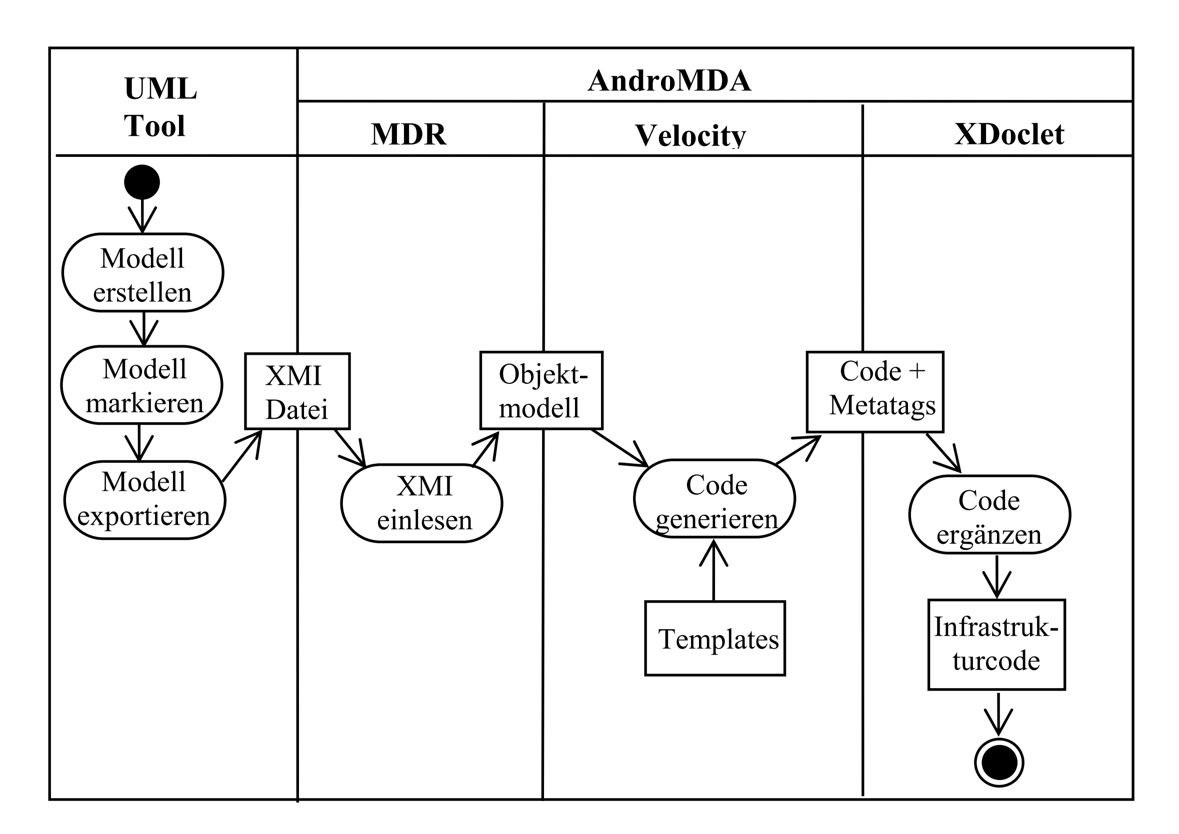


Abbildung 4.9: Ablauf eines Entwicklungsprozesses mit AndroMDA

4.5 ArcStyler

Hersteller	Interactive Objects Software GmbH
Webseite	http://www.arcstyler.de
Testversion	ArcStyler Developer Edition 4-0-108

4.5.1 Einführung

ArcStyler ist eine Softwareentwicklungsplattform, die die automatisierte Generierung von Softwareartefakten aus Modellen, basierend auf der Model Driven Architecture, ermöglicht. Entwickelt und vertrieben wird das Produkt von der deutschen Firma Interactive Objects Software GmbH, die bereits im Jahr 1990 als Consulting Unternehmen gegründet wurde. Das Hauptgeschäftsumfeld der Unternehmung war die Erstellung von Architektur, Entwurf und Implementierung von Objekt- und Komponentenorientierten Softwaresystemen. Daraus entwickelte sich ihr neues Hauptbeschäftigungsfeld, nämlich die Entwicklung eines MDA-Werkzeuges.
Entwickelt wurde dieses in Java, um nicht auf ein bestimmtes Betriebssystem beschränkt zu sein. Erkauft wird diese Unabhängigkeit durch hohe Anforderungen an die Hardware. Als minimale Konfiguration werden 1,4 GHz Prozessorleistung und 256 MB Arbeitsspeicher genannt. Doch die Praxis zeigte, dass unter diesen Vorraussetzungen ein sinnvolles Entwickeln nur bedingt durchführbar ist. Deshalb sollte man zumindest die empfohlene Konfiguration von 2 GHz Prozessorleistung und 1 GB Arbeitsspeicher erfüllen.

Die zweite Generation des ArcStylers fokussiert auf den gesamten Lebenszyklus eines Softwaresystems von Analyse, Entwurf, Entwicklung bis hin zur Verteilung und Tests. Versionen der ersten Generation (Version < 4.0) waren noch nicht mit einem integrierten Modellierungswerkzeug ausgestattet. Stattdessen wurde der Einsatz des Modellierungswerkzeugs Rational Rose von IBM empfohlen. Ab Version 4.0 wurde das UML-Modellierungswerkzeug Magic Draw der Firma No Magic im ArcStyler integriert.

Die Evaluierung wird anhand der aktuellen Version 4-0-108 des ArcStyler Developer Edition durchgeführt. Für die Beschreibung bzw. Bewertung des Tools wurden mitgelieferte Beispiele und Dokumentationen [IO-03a][IO-03b][IO-03c] herangezogen. Zusätzliche Erfahrungen aus der Fallstudie Terminverwaltung (siehe Kapitel 5),

in der ArcStyler als MDA-Werkzeug eingesetzt wird, ergänzen die durchgeführte Evaluierung.

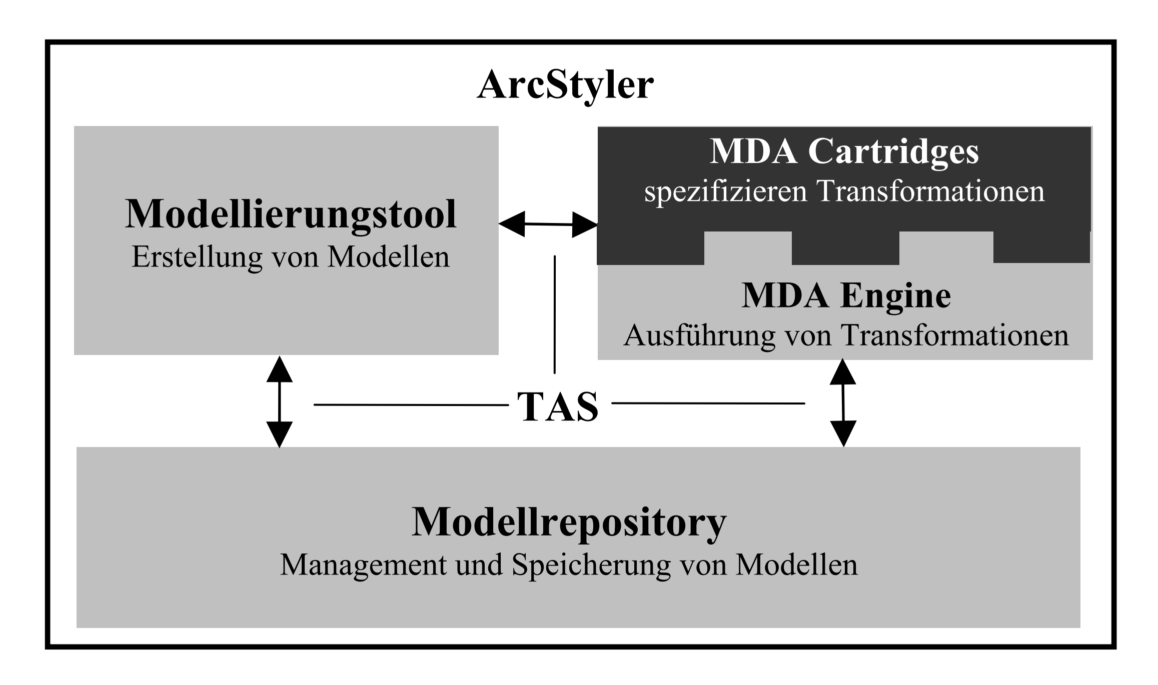


Abbildung 4.10: Architektur des ArcStyler nach [IO-03c]

Die interne Struktur des ArcStylers wird durch Abbildung 4.10 grob skizziert. Die Architektur des Werkzeuges kann nach [I0-03a] in drei Hauptmodule eingeteilt werden, welche durch den *Tool Adapter Standard* (*TAS*) verbunden sind. Die Module und TAS werden im Folgenden genauer beschrieben.

4.5.2 Modellierungswerkzeug

ArcStyler beinhaltet seit Version 4.0, wie zuvor kurz erwähnt wurde, ein integriertes Modellierungstool, welches die Modellierung von Diagrammen, basierend auf UML 1.4, ermöglicht. Abbildung 4.11 zeigt die Modellierungsperspektive des ArcStylers, wobei auf der linken Seite ein Browser dargestellt wird, der die verwendeten Modellelemente und Profiles visualisiert. Der Hauptbereich der Modellierungsschicht wird zur Erstellung von UML-Diagrammen genutzt.

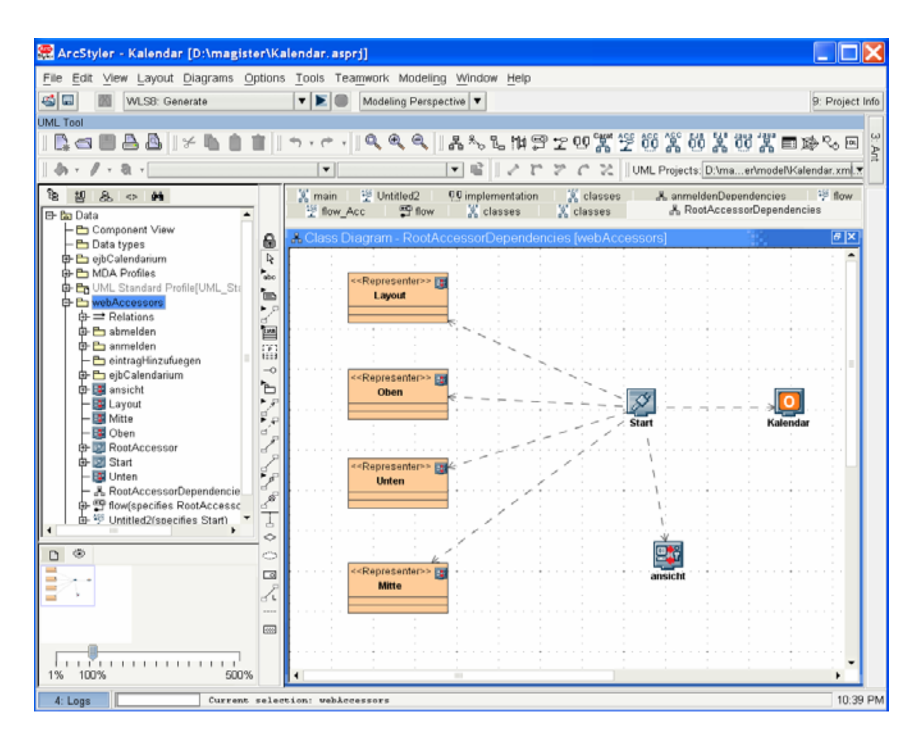


Abbildung 4.11: Modellierungsperspektive des ArcStylers

Um den Anforderungen an MDA gerecht zu werden, unterstützt das Modellierungswerkzeug Profiles. Im ArcStyler enthält ein Profile folgende Elemente:

- *Stereotype*: können zu jedem passenden Modellelement hinzugefügt werden
- *Datentypen*: können zu jedem *TypedElement* (z.B. Attribute) angegeben werden
- *Markierungen*: sind Schlüsselwort/Wert-Paare, welche in definierten Mengen gruppiert vorliegen und mit Standardwerten ausgestattet sind.

Im zweiten Kapitel dieser Arbeit wurden Markierungen besprochen und die Anforderung postuliert, dass für Markierungen intelligente Auswahloptionen vorgesehen werden sollten, um die EntwicklerIn bei der korrekten Auswahl zu unterstützen. ArcStyler realisiert diese Anforderung durch einen eigenen Menüpunkt *MDA Marks*. Abbildung 4.12 zeigt als Beispiel für Markierungen im ArcStyler ein Menü für die Markierung eines Elements für die Zielplattform Enterprise Java Beans. Zuvor wurde dieses Element mit dem Stereotyp *«ComponentSegment»* versehen, wodurch die Markierungen für EJB automatisch verfügbar werden, um die nötigen Entscheidungen über den Typ der Bean (wie EntityBean, SessionBean oder MessageBean) oder State Management (*statefull* oder *stateless*) zu treffen.

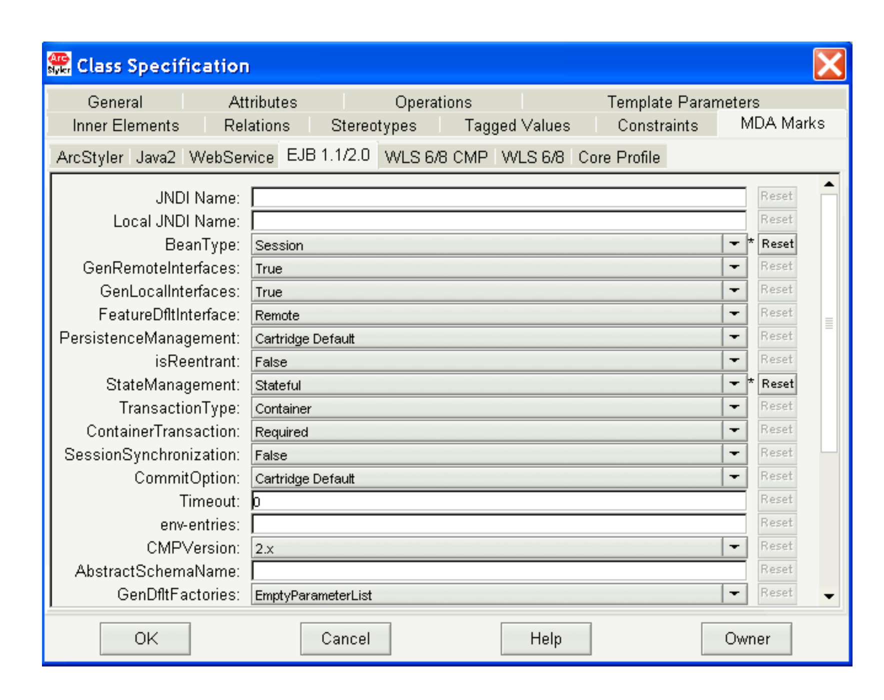


Abbildung 4.12: ArcStylers Auswahlmenü für Markierungen

Durch die beschriebene Technik ist es möglich, Modelle auf verschiedenen Ebenen (wie z.B. PIM oder PSM) zu erstellen. Durch Markierungen können Entwurfsentscheidungen bezogen auf konkrete Plattformen berücksichtigt werden, um notwendige Informationen für die Transformationen in die Modelle einzubringen.

4.5.3 MDA-Cartridges/MDA-Engine

MDA-Cartridges beinhalten Informationen über Modelltransformationen, Modellverifikationen und benötigte Profiles. Im Allgemeinen repräsentiert eine MDA-Cartridge genau eine Plattform und kapselt ihre plattformspezifischen Konzepte. Um bestimmte Plattformen im ArcStyler zu unterstützen, müssen die entsprechenden MDA-Cartridges importiert werden. Für ein Projekt können auch mehrere MDA-Cartridges gleichzeitig verwendet werden, um die Erstellung von Softwaresystemen, die auf mehreren Plattformen basieren, zu unterstützen. Tabelle 4.4 gibt einen groben Überblick über die in der Standardinstallation enthaltenen MDA-Cartridges und ihre Aufgaben.

Bezeichnung	**Beschreibung**
Java2	MDA-Cartridge für die Erstellung von Java-basierten Systemen
C#	MDA-Cartridge für die Erstellung von .NET-basierten Systemen
WebAccessor	MDA-Cartridge für JSP- oder .NET-basierte Webanwendungen
EJB 1.1	MDA-Cartridge für die Erstellung von EJB 1.1 Anwendungen.
EJB 2.0	MDA-Cartridge für die Erstellung von EJB 2.0 Anwendungen. Diese MDA-Cartridge wird weiters für die Erstellung von applikationsserverspezifischen MDA-Cartridges verwendet.
WSL 7	EJB-Cartridge für BEA WebLogic Server 7.x
WSL 8	EJB-Cartridge für BEA WebLogic Server 8.x

Tabelle 4.4: MDA-Cartridges für ArcStyler

Zu diesen mitgelieferten MDA-Cartridges können jederzeit weitere, entweder von *MDA-Cartridge Source Forge*[1] kostenlos beziehbare oder selbst entwickelte, hinzugefügt werden. Mit dem Import von neuen MDA-Cartridges kann das Modellierungswerkzeug über Profiles um neue Modellelemente erweitert werden und zusätzlich kann die MDA-Engine um neue Transformationen und Modellverifikationen erweitert werden.

Die Entwicklung von eigenen MDA-Cartridges wird durch die *Cartridge Architecture* (*CARAT*) erleichtert. Durch CARAT wird die Modellierung von MDA-Cartridges und weiters die automatische Generierung der Implementierung der MDA-Cartridges aus den Modellen ermöglicht. Somit wird auch für die Erstellung von Transformationen der MDA-Ansatz verwendet und es können Konzepte wie Wiederverwendung und Spezialisierung von MDA-Cartridges genutzt werden.

Die MDA-Engine bietet Dienste an, die von den MDA-Cartridges aufgerufen werden können, beispielsweise das Management von textbasierten Artefakten. Dabei übernimmt die MDA Engine den Schutz der manuell programmierten Teile und die Sicherstellung, dass nur geänderte Dateien neu erstellt werden.

4.5.4 Modellrepository

Die erstellten Modelle werden entweder als XMI-basierte Textdateien oder in einer verteilten und mehrbenutzerfähigen Modelldatenbank archiviert. MDA-Cartridges brauchen Zugriff auf die Modellelemente, um ihre Aufgaben wie Transformationen und Verifikationen zu erfüllen. Deshalb ist das Modellrepository über das *Java Metadata Interface* (*JMI*) [Sun02] und wegen Abwärtskompatibilität für ältere Trans-

[1] http://www.mda-at-work.com (Stand: 12.11.2004)

formationen über die *C-MOD* API zugänglich. ArcStyler verwendet ab der Version 4.0 UML 1.4 als Metamodell, doch wegen Abwärtskompatibilität wird weiterhin das proprietäre C-MOD Metamodell unterstützt. Letztgenanntes wird ab der ArcStyler Version 4.0 durch ein UML-Profile definiert, um die Erstellung von C-MOD-basierenden Modellen mit dem integrierten UML-Modellierungswerkzeug zu bewerkstelligen.

Im ArcStyler ist es nicht möglich neue Metamodelle zu definieren, es können aber neue UML-Profiles durch einen internen Profile-Editor erstellt werden. Es sei aber darauf hingewiesen, dass die fehlende Metamodellerstellungsfunktion nicht durch die Definitionsmöglichkeit von UML-Profiles ersetzt werden kann.

4.5.5 Tool Adapter Standard

Verbunden werden die drei Hauptmodule durch den ArcStyler *Tool Adapter Standard* (*TAS*). Laut [IO-03a] kann TAS als Container für Tools gesehen werden, der Services für folgende Aufgaben definiert:

- Interaktionen mit anderen Werkzeugen, die mittels TAS im ArcStyler eingebunden sind
- GUI Integration (Menüs, Toolbars, Panels)
- Selektionsmanagement von Modellelementen
- MDA-Cartridge Management
- Verbindung zwischen verschiedenen Repository Schnittstellen und Profiles
- Management von Profiles und Markierungen

TAS bietet Erweiterungspunkte, um selbstentwickelte oder von Drittanbietern erstellte Werkzeuge im ArcStyler zu integrieren. Diese integrieten Werkzeuge können die vorher definierten Services über Schnittstellen benutzen. Weitere Informationen über die TAS API und Anleitungen über die Entwicklung eigener Plugins findet man in [IO-03b].

4.6 Objecteering/UML

Hersteller	SOFTEAM
Webseite	http://www.softeam.com
Testversion	Objecteering/UML Enterprise Edition, Version 5.3.0 für Windows

4.6.1 Einführung

Das französische Unternehmen SOFTEAM betreibt bereits seit dem Jahr 1991, lange bevor der Begriff MDA definiert wurde, intensive Forschung und Entwicklung im Bereich der modellgetriebenen Softwareentwicklung. Im Jahr 1994 begann SOFTEAM in ihrem UML CASE Tool *Objecteering/UML* unter dem Begriff *Hypergenercity* [Soft01] mit der Integration von Features für modellgetriebene Softwareentwicklung. Das Konzept hinter dem Begriff Hypergenercity hatte bereits viele Gemeinsamkeiten zu dem, was heutzutage unter MDA verstanden wird. Somit war Objecteering/UML eines der ersten Tools für modellgetriebene Entwicklung am Softwaremarkt und ein Pionier für alle heute erhältlichen MDA-Werkzeuge.

Objecteering/UML entwickelte sich über die letzten zehn Jahre vom UML CASE Tool zu einem MDA Tool, das den gesamten Entwicklungsprozess von der Analyse bis zur Artefakterstellung abdeckt. Es werden Versionen für Windows, Linux und Solaris angeboten, wobei je vier unterschiedlich umfangreiche Pakete erhältlich sind. Die Spanne reicht vom einfachen UML Modellierungseditor bis hin zum mehrbenutzerfähigen MDA Tool. Für die Evaluation wird Objecteering/UML Enterprise Edition, Version 5.3.0 für Windows eingesetzt.

Als minimale Konfiguration für Projekte mit kleinen Modelldatenbanken (bis ungefähr 10 MB Speicherplatz) werden 1 GHz Prozessorleistung, 512 MB Arbeitsspeicher und 300 MB freier Festplattenspeicherplatz genannt. Die empfohlene Konfiguration für Projekte mit großen Modelldatenbanken wird mit mindestens 1 GHz Prozessorleistung, 2 GB Arbeitsspeicher und 600 MB freiem Festplattenspeicher angegeben.

Die Funktionalität von Objecteering wird durch zwei eigenständige Softwarekomponenten (siehe Abbildung 4.13) realisiert:

- *Objecteering/UML Modeler:* wird von SoftwareentwicklerInnen, wie SoftwaredesignerInnen und ProgrammiererInnen verwendet, um Softwaresysteme zu modellieren.

- *Objecteering/UML Profile Builder:* wird von SoftwarequalitätssichererInnen und SoftwarearchitektInnen eingesetzt, um UML Profile zu erstellen. Diese werden zu Modulen zusammengefasst und im Objecteering/UML Modeler verwendet.

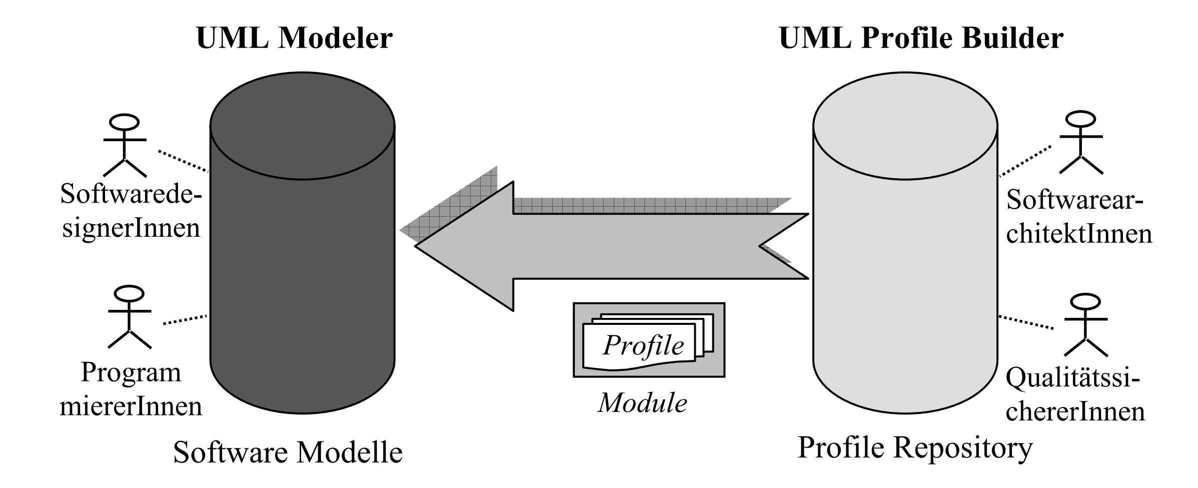


Abbildung 4.13: UML Modeler und UML Profile Builder nach [Soft99]

Die Definition und Verwendung von UML Profiles durch die mitgelieferten Tools von Objecteering/UML kann in die folgenden Schritte zusammengefasst werden:

1. Erstelle mit Hilfe des Objecteering/UML Profile Builder die benötigten Profiles für die eingesetzten Plattformen und verpacke diese zu verteilungsfähigen Modulen.
2. Verteile die erstellten Module in Modelldatenbanken.
3. Wähle im Objecteering/UML Modeler ein oder mehrere Module aus und erstelle damit Softwaremodelle.

4.6.2 Objecteering/UML Modeler

Objecteering/UML verfügt über ein eigenes UML Modellierungstool namens Objecteering/UML Modeler. Als Modellierungssprache wird bei der vorliegenden Version UML 1.4 verwendet, seitens SOFTEAM wird aber eine Umstellung auf UML 2.0 vorbereitet. Motiviert ist die Umstellung durch die bessere Unterstützung der Notation von UML Profiles in UML 2.0 und durch die Mitgliedschaft von SOFTEAM im UML 2.0 Response Team [Soft01].

Abbildung 4.14 zeigt die Hauptansicht des Objecteering/UML Modeler, welche in vier Hauptkategorien eingeteilt werden kann. Der *Explorer* (Nummer 1 in der Abbildung 4.14) zeigt in einer hierarchischen Sicht einen Baum, der alle Modellelemente eines Projektes beinhaltet. Er wird verwendet, um Elemente eines Modells (wie z.B. Klassen, Attribute und Operationen) zu erstellen und zu visualisieren. Der *Eigenschaften-Editor* (Nummer 2 in Abbildung 4.14) beinhaltet mehrere Karteikarten, die zum markierten Modellelement und dessen zugeteilten Elementen nähere Informati-

onen und Befehle anzeigen. Die *Konsole* (Nummer 3 in Abbildung 4.14) beinhaltet Feedback für die EntwicklerInnen, wie Traces, Fehlermeldungen, Warnungen und vieles mehr. Das Hauptfenster (Nummer 4 in Abbildung 4.14) zeigt das eben ausgewählte Diagramm.

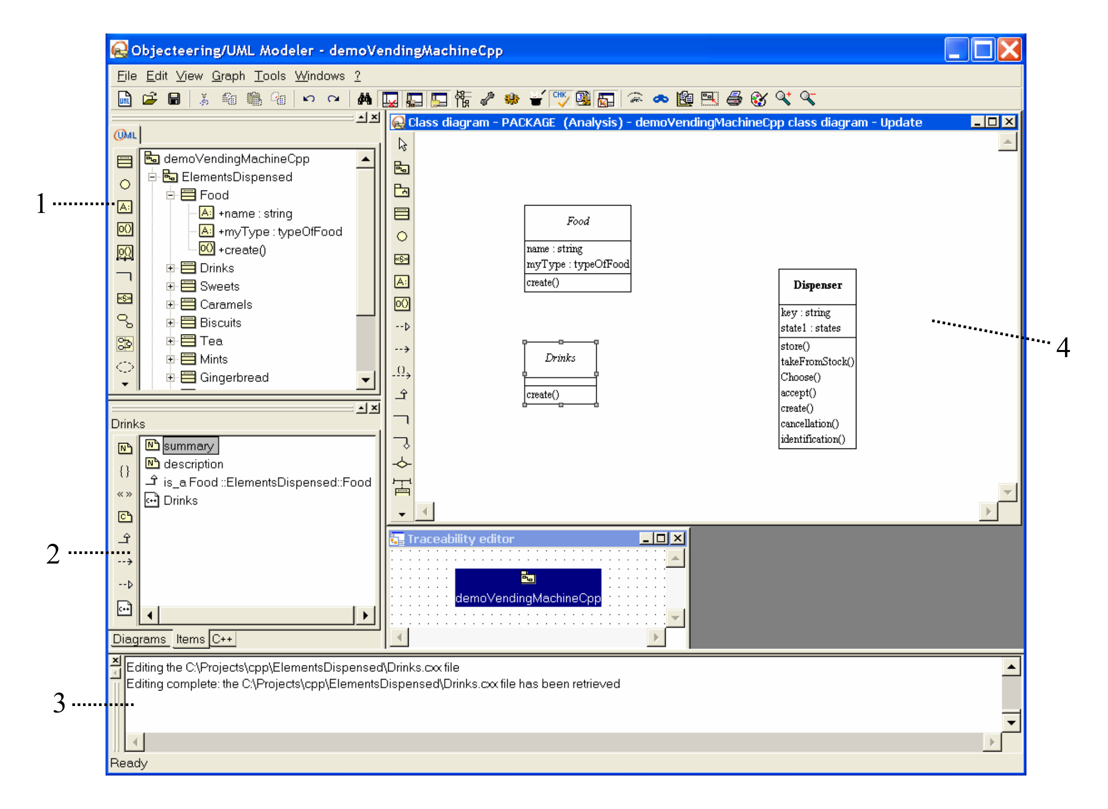


Abbildung 4.14: Hauptansicht des Objecteering/UML Modeler Tools

Solange Profiles nicht in Form von Modulen für spezielle Domänen ausgewählt wurden, stellt das Tool einen gewöhnlichen UML Modellierungseditor dar, der aber nicht nur die Syntax aller UML 1.4 Diagrammarten, sondern auch die Semantik, unterstützt. Somit können erstellte Modelle auf ihre Konsistenz überprüft werden. Werden aber Module für ein Projekt ausgewählt, so stehen zusätzlich spezifische UML Modellerweiterungen, Artefaktgenerierung, Modelltransformationen und vieles mehr zur Verfügung. Damit diese Funktionen realisierbar sind, stehen folgende Konzepte zur Verfügung, um den Objecteering/UML Modeler zu erweitern:

- *Stereotype:* Durch die ausgewählten Module werden Stereotype, die in den Profiles der Module enthalten sind, nutzbar. Zu den modellierten Elementen können diese Stereotype hinzugefügt werden, um spezifischere Informationen in das Modell einzubringen.

  Zu beachten ist, dass nur passende Stereotype einem Modellelement zuge-

wiesen werden können, d.h. die Metaklasse des Modellelementes muss mit der Basisklasse des Stereotyps übereinstimmen oder eine Spezialisierung der Basisklasse des Stereotypen darstellen. Visualisiert werden Stereotype im Objecteering/UML Modeler entweder durch die übliche Stereotypnotation mit französischen Anführungszeichen oder durch zugewiesene Icons. Als Beispiel für Stereotype könnte das Stereotyp *«Entity Bean»* der Plattform Enterprise Java Beans dienen.

- *Schlüsselwort/Wert-Paare:* Die Semantik von Modellelementen kann durch die Angabe von Schlüsselwort/Wert-Paaren noch genauer spezifiziert werden. Dabei wird berücksichtigt, welche Schlüsselwort/Wert-Paare für welche Stereotype oder Metaklassen angegeben werden dürfen. Objecteering/UML verwendet Schlüsselwort/Wert-Paare, um das Konzept von Markierungen zu realisieren. Beispielsweise könnte für eine Klasse, welche mit dem Stereotyp *«Entity Bean»* versehen ist, ein Schlüsselwort/Wert-Paar für das Management der Persistenz angegeben werden, wie *{persistenz=cmp}* für Container-verwaltete Persistenz oder *{persistenz=bmp}* für Bean-verwaltete Persistenz.

- *Einschränkungen:* Eine Einschränkung kann Restriktionen und Beziehungen ausdrücken, die mittels UML Notation nicht definierbar sind. Die Einschränkung ist in natürlicher Sprache zu formulieren, da keine semantische Auswertung der Einschränkungen im Tool vorgesehen ist. Allgemein ist die Definition von Einschränkungen für jede Instanz des Typs *ModelElement* oder davon spezialisierten Typs möglich. Mittels Profiles können aber auch eigene Einschränkungen für Metaklassen und Stereotype spezifiziert werden.

- *Notes:* In Profiles können zu Metaklassen oder Stereotype sogenannte Notes zugeordnet sein. Dadurch ist es im Objecteering/UML Modeler möglich, Notes zu den Instanzen der entsprechenden Metaklassen oder zu Instanzen von der Metaklasse spezialisierten Metaklassen hinzuzufügen. Wird ein Modellelement mit einem Stereotyp, für das Notes verfügbar sind, gekennzeichnet, so können die dem Stereotypen zugeteilten Notes zu diesem Modellelement angegeben werden. Im Objecteering/UML Modeler werden Notes mittels der Kommentarnotation aus UML visualisiert. In Notes können Textfragmente für die Generierung von Quelltexten oder Dokumentationen hinterlegt werden. Als Beispiel könnte einer Methode ein Note vom Typ JavaCode angefügt werden, die den Implementierungscode der Methode enthält.

- *Commands:* Module können über Commands verfügen, die den Objecteering/UML Modeler durch modulspezifische Popup-Menüs oder Kontext-Menüs erweitern. Diese Menüpunkte sind mit J-Methoden (siehe Kapitel 4.6.2, Abschnitt: Proprietäre Konzepte) desselben Moduls verknüpft. Das Verhalten der J-Methoden wird bei der Auswahl des Menüpunkts ausgeführt. Durch Commands können z.B. Codegenerierung, Dokumentationsgenerierung oder die Anzeige von Metriken über das Modell aufgerufen werden.

- *Work Products:* Dieses Konzept repräsentiert von Objecteering/UML generierte Artefakte wie Dokumentationen, Codedateien, Makefiles, usw. Durch Work Products ist die Konsistenz zwischen den Modellen und den erzeugten externen Artefakten gegeben. Ist für die Metaklasse eines Modellelementes ein Work Product definiert, so wird bei der Markierung dieses Modellelementes ein Symbol zur Generierung des Work Products sichtbar. Ein interner Editor (siehe Abbildung 4.15) ermöglicht die Bearbeitung der erstellten Work Products. Reicht einem die Funktionalität des internen Editors nicht aus, so ist es möglich, über Parameterspezifikationen externe Editoren für die Bearbeitung der Work Products auszuwählen.

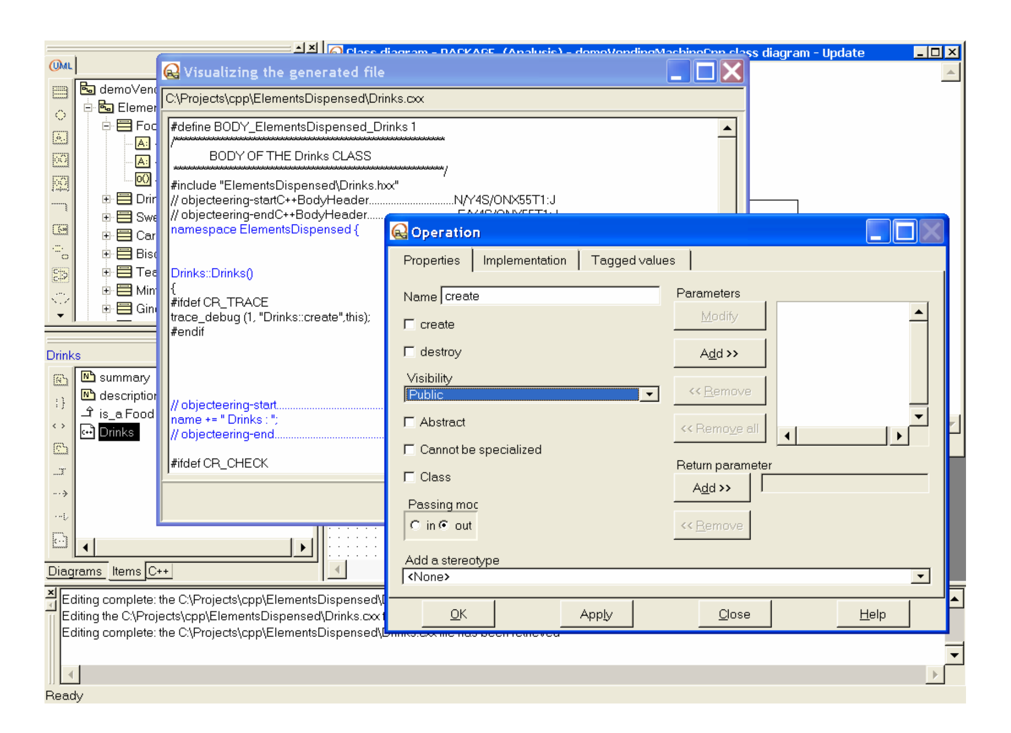


Abbildung 4.15: interner Editor des Objecteering/UML Modeler Tools

Objecteering/UML Module werden für die aktuellsten Plattformen mitgeliefert (siehe Tabelle 4.5) oder können mit Hilfe des Objecteering/UML Profile Builders er-

stellt werden. Zusätzlich sind die mitgelieferten Module durch Vererbung modifizier- bzw. spezialisierbar.

Name	**Beschreibung**
UML Profile für C++	Ermöglicht die Verwendung von Design Patterns und Codegenerierung für C++
UML Profile für Java	Ermöglicht Java Codegenerierung, Reverse Engineering, Konsistenz zwischen generiertem Code und Modell und Design Patterns für Java
UML Profile für SQL	Transformiert ein UML Modell in ein relationales Datenbankmodell und ermöglicht SQL Codegenerierung
UML Profile für EBJ	Transformiert ein UML Modell in ein EJB spezifisches Modell, ermöglicht automatische Codegenerierung (inklusive Deployment Informationen) und Reverse Engineering; bietet erweiterte Profiles für die aktuellsten Applikationsserver wie z.B. Websphere und Weblogic
UML Profile für VB	Ermöglicht VB Codegenerierung von UML Modellen
UML Profile für XMI	Ermöglicht den Export und Import von Modellen durch XMI
UML Profile für CORBA	Ermöglicht die Generierung von komplettem CORBA IDL Code
UML Profile für Test	Ermöglicht die Modellierung von Testfällen. Weiters wird ein „Tests für Java“ Subprofile geboten, das den gesamten Java Code für die benötigten Tests mittels JUnit generiert.
UML Profile für Reverse COM	Erstellt aus COM Schnittstellen automatisch UML Modelle
UML Profile für Dokumentationsgenerierung	Generiert Dokumentationen als Microsoft Word oder HTML Dokumente; durch einen Dokumentationsschablonen Editor können neue Schablonen erstellt werden.
UML Profile für Design Patterns	Automatisiert die Verwendung von Entwurfmuster durch Modelltransformationen
Metrics	Ermöglicht die quantitative und qualitative Auswertung von Modellen auf Grundlage bekannter objektorientierter Metriken
UML Profile für CMS	Bietet Subprofiles für ClearCase, CVS und viele mehr
UML Profile für Requi-	Ermöglicht eine ausführliche Anforderungsanalyse mit-

rementsanalyse	tels UML und die Traceability zwischen Anwendungsfällen und weiteren Modell- bzw. Codeteilen
UML Profile für SPEM	Ermöglicht die Modellierung von Softwareerstellungsprozessen durch das von der OMG standardisierte Software Process Engineering Management Profile
UML Profile für EDOC	Ermöglicht die Modellierung von Enterprise Distributed Object Computing, wird aber erst mit der Einführung von UML 2.0 in Objecteering/UML erhältlich sein

Tabelle 4.5: mitgelieferte Module von Objecteering/UML

4.6.3 Objecteering/UML Profile Builder

Das Objecteering/UML Profile Builder Tool wird verwendet, um UML Profiles für den Einsatz im Objecteering/UML Modeler zu entwerfen und zu implementieren. UML Profiles enthalten Elemente, welche die plattformspezifische Modellierung, Modelltransformationen, Artefaktgenerierung und die Validierung von erstellten Modellen im Objecteering/UML Modeler ermöglichen. Um den Einsatz der Profiles im Objecteering/UML Modeler zu gewährleisten, müssen diese zuerst zu Modulen gepackt werden.

Die mit Objecteering/UML mitgelieferten Module (siehe Tabelle 4.5) wurden mit Objecteering/UML Profile Builder erstellt. Diese Module und ihre zugehörigen Profiles können zwar nicht direkt verändert werden, es können aber neue Profiles von diesen abgeleitet und dadurch diese Profiles für individuelle Anforderungen erweitert werden.

Die Hauptansicht des Objecteering/UML Profile Builder, die in drei Bereiche gegliedert ist, wird in Abbildung 4.16 dargestellt. Der *Meta-Explorer* (Nummer 1 in Abbildung 4.16) zeigt eine Baumstruktur, die alle Elemente eines Projektes enthält. Der *Meta-Eigenschaften-Editor* (Nummer 2 in Abbildung 4.16) zeigt die Eigenschaften des aktuell markierten Elementes. Die *Konsole* (Nummer 3 in Abbildung 4.16) beinhaltet wie beim Objecteering/UML Modeler Tool Feedback für die EntwicklerIn. Das Hauptfenster (Nummer 4 in Abbildung 4.16) wird zum Testen von erstellten Profiles verwendet, d.h. es ist nicht nötig, in den Objecteering/UML Modeler zu wechseln, um die Funktionalität des gerade erstellten Profile zu testen.

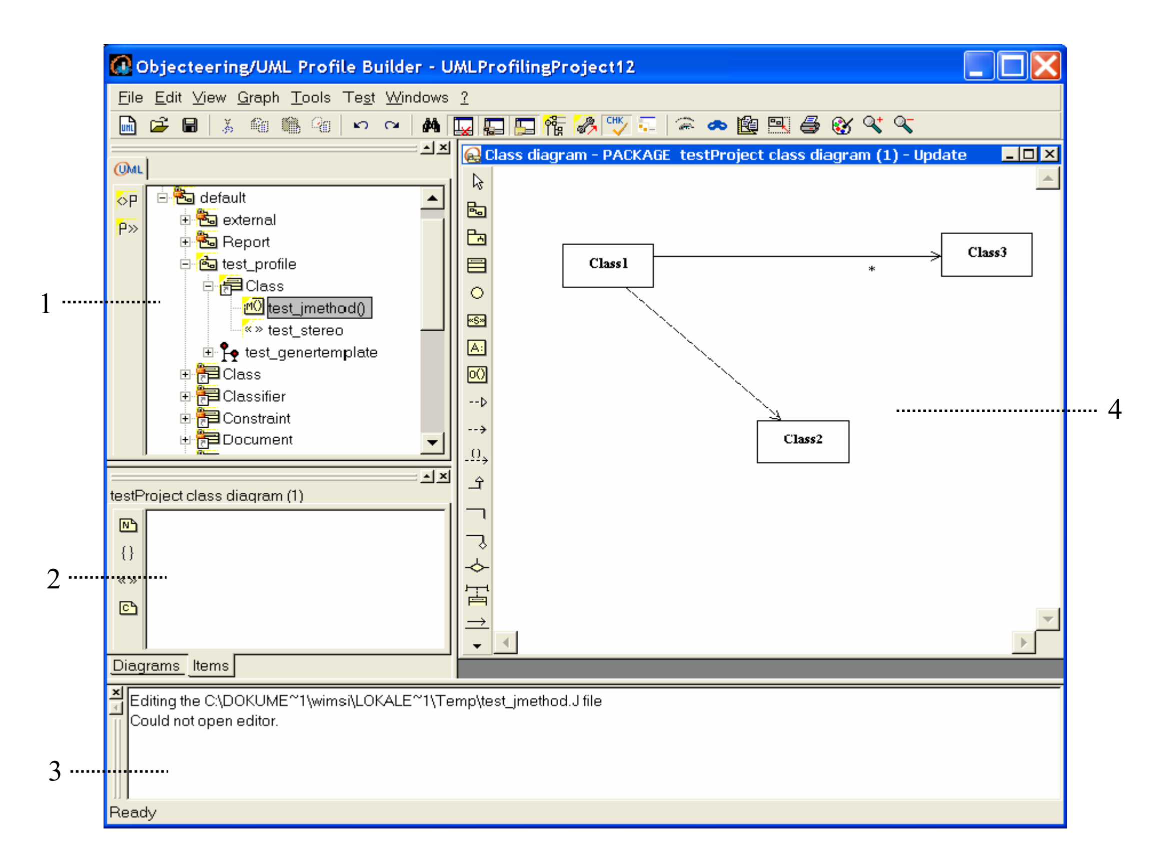


Abbildung 4.16: Hauptansicht des Objecteering/UML Profile Builders

SOFTEAM fügte Konzepte für dynamisches Verhalten zu standardkonformen UML Profiles hinzu, um die Anforderungen an MDA Kompabilität zu erfüllen. Ein Großteil der Spezifikation des dynamischen Verhaltens wird durch eine proprietäre Sprache, genannt J, realisiert. Diese Sprache bietet Unterstützung für die Erstellung von Profiles und die dafür notwendigen Arbeiten auf Metamodellebene.

Die Sprache J

J ist eine Sprache die die Erstellung von UML Profiles im Objecteering/UML Profile Builder unterstützt und die Parametrisierung und Steuerung des Objecteering/UML Modeler erlaubt. Die Sprache J ist eine objektorientierte Sprache, deren Syntax ähnlich zu der von Java ist. J ist zur Arbeit auf Metamodellebene und zur Erstellung von UML Profiles konzipiert. Da J auf dem Objecteering/UML-Metamodell (entspricht dem UML 1.4 Metamodell) basiert, bietet diese Sprache spezifische Konzepte, um die Verarbeitung der erstellten Modelle und die Navigation zu erleichtern. J stellt eine interpretierbare Programmiersprache dar. Dadurch können Codefragmente im Objecteering/UML Profile Builder rasch modifiziert und getestet werden.

J wird auf Metamodellebene ausgeführt und hantiert mit Modellelementen, welche durch BenutzerInnen des Objecteering/UML Modeler erstellt werden. In J sind zwei

Arten von vordefinierten Klassen verfügbar: einersteits Klassen für grundlegende Datentypen (wie z.B.: *String, Float, Integer*) und andererseits solche, die Elemente des Metamodells von Objecteering/UML repräsentieren (wie z.B. *Classifier, Attribute, Package*).

Bei der Erstellung von J-Programmen werden nur vorhandene Klassen genutzt, d.h. J- ProgrammierInnen können entweder neue Methoden für die vorhandenen Klassen definieren oder vorhandene redefinieren, aber können selbst keine neuen Klassen deklarieren. J wird also verwendet, um Methoden auf Metaklassenebene (Instanzen des UML Metamodells) zu definieren.

Wie zuvor besprochen, kann mittels J auf das Metamodell von Objecteering/UML zugegriffen werden. J verwendet *Metaklassen, Metaassoziationen, Metarollen* und *Metaattribute*, um durch ein Modell navigieren zu können und damit die benötigten Modellinformationen zu bekommen.

Die Sprache J baut sehr stark auf das Konzept *Nachrichten* auf. In Objecteering/UML wird eine Nachricht allgemein als Aufruf, der ein mögliches Ergebnis liefert, definiert. Meistens beschränkt sich aber eine Nachricht auf die Abfrage eines Attributwertes oder den Aufruf einer J-Methode. Um Nachrichten in J effizient zu nutzen, werden zwei Hauptkontrollkonstrukte angeboten, nämlich das Versenden von Nachrichten (ähnlich einem Methodenaufruf in Java) und der *Diffusionsmechanismus*. Mit Hilfe des Diffusionsmechanismus sind Mengen von Modellelementen sehr einfach zu verarbeiten, ohne dabei auf den Gebrauch von komplizierten Iteratoren oder for-Schleifen zurückgreifen zu müssen. Wird die Diffusion auf eine nicht leere Elementmenge angewandt, so wird für jedes Element die angegebene Nachricht versendet. Wendet man jedoch die Diffusion auf eine leere Menge an, wird eine Nulloperation (noop) ausgeführt. Syntaktisch wird die Diffusion als .< repräsentiert.

Weiters wird der Diffusionsmechanismus durch zwei spezielle Operatoren unterstützt. Mit Hilfe des *select*-Operators werden bestimmte Elemente aus einer Menge aufgrund eines boolschen Ausdrucks gefiltert. Das Ergebnis ist eine Teilmenge, in der alle Elemente die Bedingung erfüllen. Der *while*-Operator wird eingesetzt, um eine Menge von Elementen so lange zu durchlaufen, bis ein angegebener boolscher Ausdruck nicht mehr erfüllt wird. Sobald die Bedingung nicht erfüllt ist, bricht die Verarbeitung der Diffusion ab.

Anonyme Methoden sind ein Spezialfall der Diffusion und werden als Methoden ohne Namen und Parameter angeschrieben. Es wird zuerst eine Referenz auf eine Elementmenge angegeben und darunter ein Programmblock in geschwungenen Klammern angefügt. Bei der Ausführung des Programms wird der gesamte Programmblock für jedes einzelne Element der Menge durchgeführt.

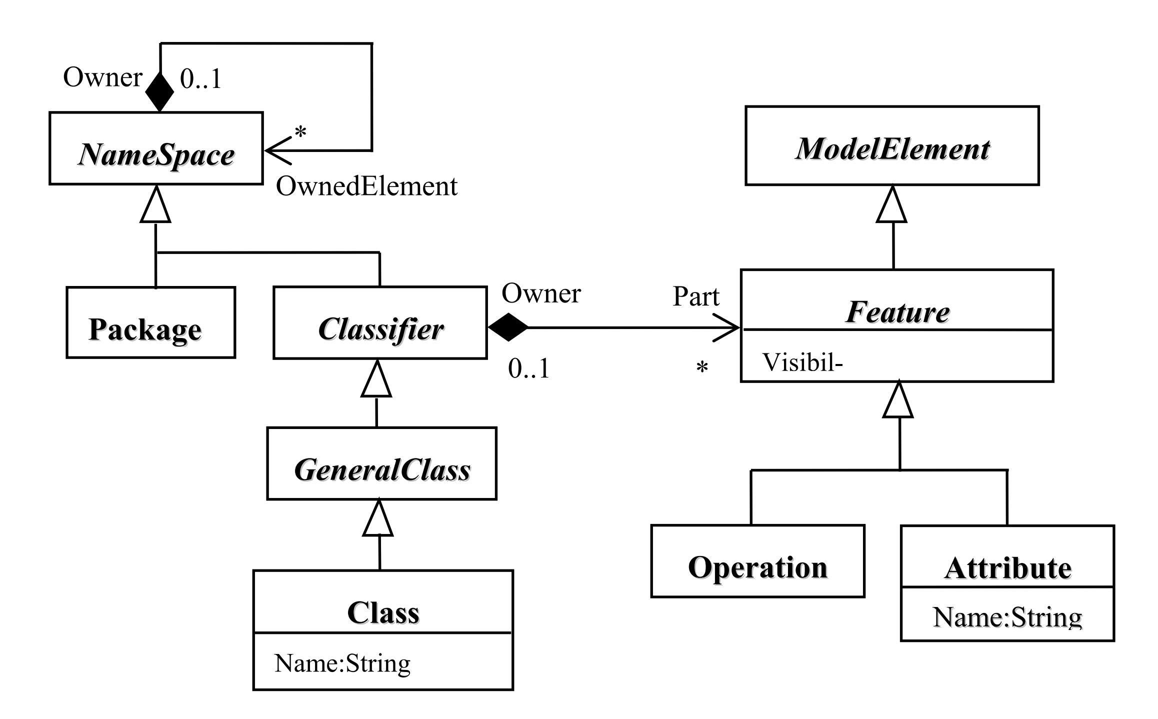


Abbildung 4.17: Auszug aus dem Objecteering/UML Metamodell

Das folgende J Programm ist ein Beispiel für den Einsatz des Diffusionsmechansimus. Die Aufgabe dieses Programms ist die Ausgabe aller öffentlichen Attribute der Klassen eines Packages. Der Zusammenhang zwischen Packages, Klassen und Attributen auf Metamodellebene wird in Abbildung 4.17 gezeigt. Für die Erstellung des Programms ist die Kenntnis des Metamodells von großer Bedeutung, da die Metamodellelemente im Programmcode korrekt angesprochen werden müssen. Die Methode listAttributes() wird zur Metaklasse Package hinzugefügt. Auf Metamodellebene besitzt ein Package Elemente vom Typ *NameSpace* (siehe in Abbildung 4.17 die Rolle *OwnedElement*), da die Klasse *Package* von der Klasse *NameSpace* abgeleitet wird. Mit `OwnedElementClass` können nun alle Klassen aus allen Elementen eines Packages herausgefiltert werden. Die vordefinierte Hilfsmethode `StdOut.write(String s)` gibt den übergebenen String über die Konsole aus. Danach wird eine Diffusion für alle Attribute einer Klasse mit Hilfe einer anonymen Methode angewendet, wobei noch zusätzlich ein *select*-Operator angegeben wird, um nur öffentliche Attribute zu berücksichtigen. In dem darunter liegenden Pro-

grammblock werden die Namen der öffentlichen Attribute über die Konsole ausgegeben.

```
Package: listAttributes()
{
    OwnedElementClass
    {
        StdOut.write(„Class:“, Name, NL);
        PartAttribute.<select(Visibility == Public)
        {

            StdOut.write(NL, “Public attribute:”, Name);
        }
    }

}
```

Wie das Codebeispiel demonstriert, bedarf es natürlich an Wissen über das in Objecteering/UML verwendete Metamodell. Im Metamodel User Guide [Soft04d] wird deshalb eine detaillierte Beschreibung des Metamodells geboten. Weitere J-Programmbeispiele und eine ausführliche Einführung in die Sprache J findet man im J Language User Guide [Soft04b]. In Objecteering/UML werden über J-Libraries zahlreiche Funktionen wie Sessionmechanismus für Modelltransformationen, Erzeugung von neuen User Interfaces oder Unterstützung für Arbeiten mit dem Dateisystem geboten. Eine ausführliche Einführung über angebotene J-Libraries erhält man im J-Libraries User Guide [Soft04d].

Der Objecteering/UML Profile Builder bietet der BenutzerIn zahlreiche Konzepte, die für die Erstellung von Profiles genutzt werden können. Die verfügbaren Konzepte können einerseits in UML standardkonforme Konzepte (wie z.B.: Stereotype, Schlüsselwort/Wert-Paare) und andererseits in von Objecteering/UML eigens definierte, proprietäre Konzepte (wie z.B.: J-Methoden, Commands) eingeteilt werden. Im Folgenden sollen zuerst die UML standardkonformen Konzepte und danach die proprietären Konzepte aufgelistet und erläutert werden.

UML konforme Konzepte

- *Referenzen auf Metaklassen:* Referenzen auf Metaklassen sind Referenzen auf UML Metamodellelementen wie z.B. Classifier, Class oder Package. Um Metaklassen durch ein Profile zu erweitern, müssen dafür im Profile Referenzen auf die zu erweiternden Metaklassen definiert werden. Diese Referenzen werden explizit modelliert bzw. im Meta-Explorer angezeigt und sind der Ausgangspunkt für alle möglichen Erweiterungsmöglichkeiten wie z.B. Ste-

reotype oder J-Methoden. Die Erweiterungen werden der Referenz auf die Metaklasse zugewiesen. Um die Zugehörigkeit der Erweiterungen zu den entsprechenden Metaklassen zu gewährleisten, verfügen alle Erweiterungsmöglichkeiten über ein Attribut *baseClass* vom Typ String, das die erweiterte Metaklasse repräsentiert. Eine Metaklasse ist über Referenzen durch beliebig viele verschiedene Profiles erweiterbar.

- *Stereotype:* Stereotype werden verwendet, um Metaklassen für bestimmte Domänen zu spezialisieren. Dies wird im Objecteering/UML Profile Builder durch das Anfügen von Stereotypen zu Referenzen auf Metaklassen ermöglicht. Um Stereotype mit einer eigenen grafischen Notation auszustatten, werden Icons (Bilddateien im bmp- oder gif-Format) angegeben, die anstatt der Repräsentation der referenzierten Metaklasse angezeigt werden.

- *Schlüsselwort/Wert-Paare:* Das Objecteering/UML Profile Builder Tool wird verwendet, um Typen von Schlüsselwort/Wert-Paaren für Stereotype oder Referenzen auf Metaklassen zu modellieren. Die optionale Angabe von Parametern (Standardwert ist NULL) für Schlüsselwort/Wert-Paare ist erlaubt. Werden keine Parameter definiert, so sind Schlüsselwort/Wert-Paare nur für die Markierung einer erfüllten Eigenschaft (z.B. persistent) eines Elementes einsetzbar. Die Parameter sind aber nicht typisierbar, denn bis auf den Standardwert NULL werden Parameter generell als Strings verarbeitet. In Objecteering/UML wird das von [OMG02a] geforderte Markierungskonzept, um die für die Transformationen notwendigen Entscheidungsalternativen zu bestimmen, durch die Definitionsmöglichkeit von Schlüsselwort/Wert-Paartypen umgesetzt.

- *Einschränkungen:* Einschränkungen werden zu Referenzen auf Metaklassen oder zu Stereotypen angefügt. Eine Einschränkung ist durch ihren Namen identifizierbar und beinhaltet einen beliebigen Text, der die Einschränkung spezifiziert. Leider wird weder OCL noch eine andere Einschränkungssprache von Objecteering/UML unterstützt. Dadurch werden Einschränkungen semantisch nicht ausgewertet und können für Transformationen nur beschränkt genützt werden.

Proprietäre Konzepte

- *Notes:* Um bestimmten Typen von Modellelementen zusätzliche Textfragmente für Kommentare oder Codeteile zuweisen zu können, ist es möglich, neue Typen von Notes anzugeben. Typen von Notes werden entweder zu Referenzen auf Metaklassen oder Stereotypen im Meta-Explorer angefügt. In Instanzen von Notes-Typen wird der gewünschte Text, der weiters für die Artefaktgenerierung genutzt wird, eingetragen. Da unterschiedliche Typen von Notes erstellt werden, können diese aufgrund ihres identifizierbaren Typs für verschiedene Aufgaben eingesetzt werden.

- *J-Methoden:* J-Methoden werden für Referenzen auf Metaklassen definiert und dadurch einer Metaklasse zugeordnet. Wie der Name dieses Konzeptes vermuten lässt, werden J-Methoden in der Sprache J geschrieben. J-Methoden werden während der Modellierung durch das Objecteering/UML Modeler Tool intern oder direkt von der BenutzerIn durch die Aktivierung eines Commands über das Kontextmenü aufgerufen. J-Code kann entweder durch einen im Objecteering/UML Profile Builder internen Editor oder durch einen beliebigen externen Texteditor erstellt werden. Die durch das Objecteering/UML Metamodell vordefinierten public oder protected J-Methoden der referenzierten Metaklasse oder deren Eltern können redefiniert werden.

- *J-Attribute:* Im Objecteering/UML Profile Builder ist es möglich, Metaattribute für J-Klassen in Form von J-Attributen zu definieren. In der evaluierten Version werden nur Klassenattribute als Metaattribute unterstützt, da Instanzattribute durch das Objecteering/UML Metamodell fix vorgegeben sind. J-Attribute werden von J-Methoden derselben Klasse verwendet und sind nicht persistent, d.h. die Werte von J-Attributen werden nicht im Modell gespeichert.

- *Modulparameter:* Um die Konfiguration der Profiles durch BenutzerInnen zu erlauben, sind Profiles durch Modulparameter parametrisierbar. Die BenutzerIn kann auf bereits definierte Modulparameter über das „Module Configuration“ Fenster zugreifen und so Optionen für das Modul auswählen. Zusätzlich kann in J-Methoden auf die Werte der Modulparameter zugegriffen werden. Dadurch wird die Berücksichtigung von individuellen Einstellungen für Artefaktgenerierung, Modelltransformationen und vieles mehr möglich.

- *Work Products:* Work Products repräsentieren Ergebnisse (meistens externe Dateien) des Generierungsprozessses, die im Objecteering/UML Modeler aufgerufen werden. Als Beispiele für Work Products können Java Quelltexte, SQL Skripte oder Dokumentationen angegeben werden. Zu einem Work Product können beliebig viele Metaklassen assoziiert werden, über deren Instanzen das Work Product erstellt werden kann. Wird eine Instanz einer Metaklasse, welche über Work Products verfügt, markiert, erscheint ein Icon für die automatische Erstellung des Work Products. Mit Hilfe des Objecteering/UML Profile Builder Tools sind neue Typen von Work Products erstellbar. Zu einem Work Product können Metaattribute vom Typ String oder Boolean hinzugefügt werden. Bei der Erstellung eines Work Products im Objecteering/UML Modeler werden die Metaattribute automatisch angezeigt und sind von der BenutzerIn veränderbar. Dadurch können die Pfade und Dateinamen der zu generierenden Artefakte einfach spezifiziert werden.

- *Module:* Ein Modul ist eine kompakte Einheit von beliebig vielen Profiles und Commands. Module werden im Objecteering/UML Profile Builder erstellt und in prof-Dateien gepackt. Die gepackten Module werden mit Hilfe des mitgelieferten Administrationstools in Modelldatenbanken importiert. Nähere Informationen zu der Administration von Modulen siehe [Soft04g]. Die in den Modelldatenbanken gelagerten Module können im Objecteering/UML Modeler genutzt werden. Ein Modul darf durch beliebig viele andere Module spezialisiert werden. Einzige Ausnahme ist das Verbot von Vererbungskreisen, d.h. ein Profil darf nicht von sich selbst erben. Von einem Elternmodul werden alle Modulparameter und alle enthaltenen Profiles geerbt.

- *Commands:* Um den BenutzerInnen des Objecteering/UML Modeler den Zugriff auf J-Methoden zu ermöglichen, werden zu Modulen Commands angegeben. Bei Commands handelt es sich um Befehle, die über Kontextmenüs von bestimmten Modellelementen aufgerufen werden. Wird ein Befehl aktiviert, folgt daraus die Ausführung der zugeteilten J-Methode. Da eine J-Methode direkt mit einer Metaklasse verbunden ist, steht ein Command auch mit einer Metaklasse in Beziehung. Eine über einen Command ausführbare J-Methode darf keine Parameter besitzen und muss zusätzlich public sein.

- *Document Templates und Generation Templates:* Die Konzepte Document Templates und Generation Templates beschreiben die zu erzeugenden Arte-

fakte durch die Definition einer hierarchischen Teststruktur. Templates sollen den Aufwand der J-Programmierung für Artefaktgenerierung drastisch reduzieren, denn jede Zielplattform, die ein Textformat besitzt (wie z.B. Java, XMI, C++, usw.), ist durch die Definition von Templates beschreibbar. Die Generierung von Dokumentationen stellt einen speziellen Fall von Artefaktgenerierung dar, denn durch spezielle Formate wie z.B. PDF, HTML und die Einbindung von Grafiken der Modelle müssen erweiterte Anforderungen im Vergleich zu gewöhnlicher Textgenerierung berücksichtigt werden. Um weiterführende Informationen zur Erstellung von Document Templates zu erhalten, wird auf [Soft04f] verwiesen.

4.7 Evaluierung der Werkzeuge

Für die Evaluierung werden Werkzeuge der Kategorien Executeable UML, plattformspezifische und eigentliche MDA-Werkzeuge, welche in den Kapitel 5.1 bis 5.6 beschrieben wurden, herangezogen. Um der LeserIn einen raschen und aussagekräftigen Überblick über die Evaluierung der MDA-Werkeuge zu verschaffen, werden diese in tabellierter Form gegenübergestellt. Dabei repräsentieren die Zeilen die Anforderungen und die Spalten die evaluierten Werkzeuge. Die Produkte werden anhand des im Kapitel 3.2 definierten Kriterienkatalogs mit folgender Skala bewertet:

+	Kriterium wird vollständig umgesetzt
O	Kriterium wird nur teilweise umgesetzt
-	Kriterium wird nicht umgesetzt
?	Kriterium kann wegen fehlender Informationen nicht bewertet werden

Um die Evaluierung übersichtlich darstellen zu können, werden folgende Abkürzungen für die vorher beschriebenen Werkzeuge eingeführt:

NU	Nucleus Bridgepoint
IU	iUML/iCCG
OP	OptimalJ
AN	AndroMDA
AR	ArcStyler
OB	Objecteering/UML

Im Folgenden wird je Kriterienkategorie eine Tabelle angegeben, um die Ergebnisse der Toolevaluation zusammenzufassen.

Modellierungskriterien	**NU**	**IU**	**OP**	**AN**	**AR**	**OB**
M1: Unterstützung von UML 1.x / 2.0	O	O	+	+	+	+
M2: Erstellung von UML Profiles	-	-	-	O	+	+
M3: Definition von auf MOF basierenden Meta-modellen	-	-	+	-	-	-
M4: Repository für (Meta-)Modelle	+	+	+	-	+	+
M5: OCL Unterstützung	-	-	-	O	+	-
M6: Modellvalidierung	+	+	-	-	+	O
M7: Modellsimulation	+	+	-	-	+	-
M8: Unterstützung von Markierungen	+	+	+	+	+	+
M9: User Interface Modellierung	-	-	+	O	+	-
M10: Unterstützung von abstrakten/konkreten Plattformen	-	-	+	O	O	O
M11: Unterstützung von Action Semantic	+	+	-	-	-	-
M12: Anforderungsanalyse	O	O	-	O	O	+
M13: Modellierung von Testfällen	O	O	O	-	+	+
M14: Mehrbenutzerfähigkeit	+	+	+	-	+	+

Tabelle 4.6: Modellierungsanforderungen

Modelltransformationskriterien	**NU**	**IU**	**OP**	**AN**	**AR**	**OB**
T1: Modell-zu-Modell Transformation	-	-	+	-	-	-
T2: Modell-zu-Code Transformation	+	+	+	+	+	+
T3: Traceablitity zwischen Quell- und Zielelemen-ten	-	-	-	-	-	+
T4: Modifikation/Erstellung von Transformationen	+	+	+	+	+	+
T5: Debugging Informationen	-	-	+	-	O	-
T6: Modellierung von Transformationen	-	-	-	-	+	-

Tabelle 4.7: Modelltransformationsanforderungen

Artefaktgenerierungskriterien	**NU**	**IU**	**OP**	**AN**	**AR**	**OB**
A1: geschützte Bereiche	-	-	+	O	+	O
A2: Debugging Informationen für Artefaktgenerierung	+	+	+	O	+	+
A3: hohe Codequalität	+	+	+	+	+	+
A4: automatische Dokumentationserstellung	O	O	O	-	O	+

Tabelle 4.8: Artefaktgenerierungsanforderungen

Toolinteroperabilität-/Toolintegrationkriterien	**NU**	**IU**	**OP**	**AN**	**AR**	**OB**
I1: Unterstützung von XMI	+	-	+	O	+	+
I2: Erweiterung	-	-	-	O	+	-
I3: Unterstützung von Programmier-IDEs	-	-	O	-	+	O

Tabelle 4.9: Toolinteroperabilitäts-/Toolintegrationsanforderungen

ökonomische Kriterien	**NU**	**IU**	**OP**	**AN**	**AR**	**OB**
O1: Anschaffungskosten	?	?	?	+	-	?
O2: unterschiedliche Versionen	O	O	+	-	+	+
O3: angebotene Schulungen	+	+	O	-	+	+
O4: Werkzeugdokumentation	O	O	+	O	+	+
O5: Supportmöglichkeit	O	O	+	-	+	+
O6: Benutzerfreundlichkeit	O	O	O	-	+	+

Tabelle 4.10: ökonomische Anforderungen

Kapitel 5

Fallstudie

Dieses Kapitel beschreibt eine Fallstudie, in der eine mehrschichtige Webapplikation anhand des MDA-Ansatzes implementiert wird. Dabei besitzt die Webapplikation nur eine minimale Funktionalität, da der Fokus der Fallstudie nicht auf der Erstellung eines komplexen Systems liegt, sondern es soll je ein Beispiel für die Implementierung eines Konzeptes, wie z.B. Datenbankkommunikation, Benutzerschnittstelle und Geschäftslogik, geboten werden. Durch die Fallstudie wird gezeigt, welche Tätigkeiten in einem MDA-Entwicklungsprozess für die EntwicklerInnen anfallen und welche durch Werkzeugeinsatz effizient und automatisiert durchgeführt werden können. Weiters sollen Vorteile und auch mögliche Nachteile eines modellgetriebenen Entwicklungsansatzes aufgezeigt werden. Im Zuge der Erstellung der Anwendung soll auch der derzeitige Stand der MDA-Werkzeuge durch den Einsatz des ArcStyler Werkzeuges beispielhaft aufgezeigt werden.

Dieses Kapitel ist in vier Hauptbereiche eingeteilt. Unterkapitel 5.1 beschreibt die Anforderungen an das für die Fallstudie zu erstellende System. Der Abschnitt 5.2 gibt eine kurze Einführung in die einzusetzenden Technologien. Die Durchführung der konkreten Entwicklung des Systems wird im Abschnitt 5.3 beschrieben. Abschließend werden die Erfahrungen aus der Fallstudie im Unterkapitel 5.4 zusammengefasst. Das vollständige Modell der Terminverwaltung befindet sich in Anhang A und der Code der Implementierung befindet sich in Anhang B.

5.1 Beschreibung der Terminverwaltung

Als Fallstudie wird eine Terminverwaltung als Webapplikation implementiert. Die Terminverwaltung ist ein beispielhafter Auszug aus dem CALENDARIUM-Projekt [Kapp03].

Benutzer der Terminverwaltung sollen über eine Website Termine anlegen, bearbeiten und löschen können. Vorraussetzung dafür ist eine erfolgreiche Authentifizierung der Benutzer. Diese funktionalen Anforderungen reichen bereits aus, um die wichtigsten Aspekte von Webanwendungen für die Fallstudie zu berücksichtigen. Das Anwendungsfalldiagramm in Abbildung 5.1 fasst die funktionalen Anforderungen an die Webapplikation zusammen.

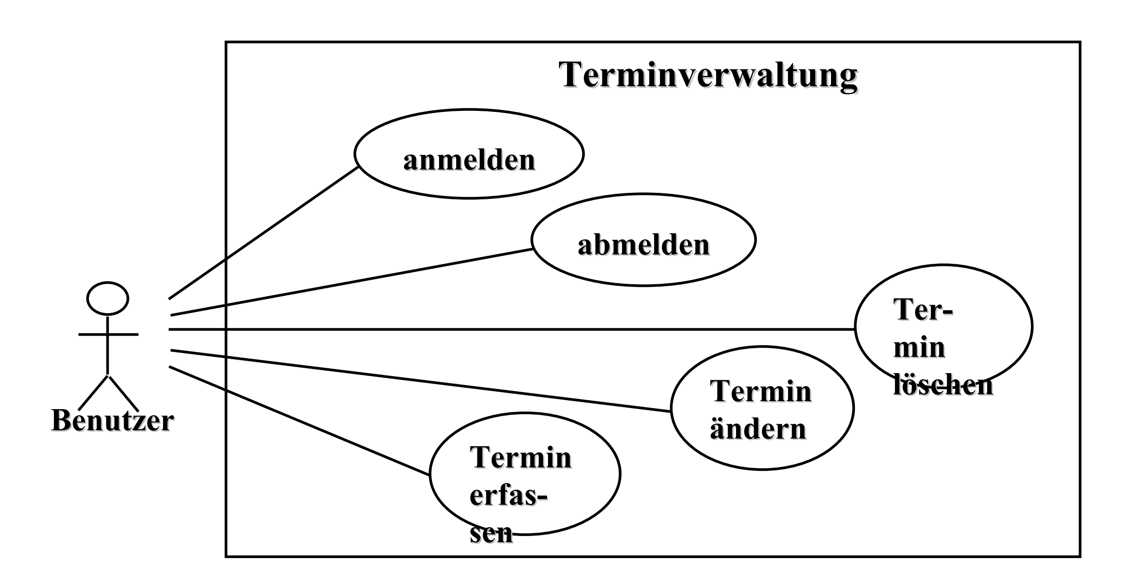


Abbildung 5.1: funktionale Anforderungen an die Terminverwaltung

Jeder Benutzer wird durch einen eindeutigen Benutzernamen identifiziert und besitzt zusätzlich ein Kennwort, um sich für die Anwendung zu authentifizieren. Ein Benutzer kann beliebig viele Termine besitzen. Ein Termin besteht aus Terminnamen, Beginndatum, Beginnzeit, Dauer und Ort. Das Klassendiagramm in Abbildung 5.2 gibt einen Überblick über die wichtigsten Entitäten der Terminverwaltung.



Abbildung 5.2: Entitäten der Terminverwaltung

5.2 Einzusetzende Technologien

Für die technische Realisierung der Terminverwaltung ist der Einsatz der *J2EE* Plattform [SUN03a], insbesondere der von *Enterprise Java Beans* (*EJB*) [SUN03b], gefordert. Die Architektur der Anwendung soll anhand des MVC-Musters entworfen werden. Im Folgenden werden Enterprise Java Beans näher erläutert.

Enterprise Java Beans

EJBs sind serverseitige Softwarekomponenten, die in einer verteilten und mehrschichtigen Umgebung, genannt *EJB Container*, ablaufen. Eine EJB kann aus einem Java-Objekt oder aus mehreren Java-Objekten bestehen, da eine Komponente eine größere Einheit als ein einzelnes Objekt darstellt. Unabhängig davon, wie eine EJB aufgebaut ist, kann ein Client über eine einzige Komponentenschnittstelle mit der EJB kommunizieren. Diese Schnittstelle sowie die EJB selbst müssen mit der EJB Spezifikation [SUN03b] übereinstimmen. Die EJB Spezifikation verlangt, dass EJBs vorgeschriebene Methoden implementieren. Diese vorgeschriebenen Methoden erlauben den EJB Containern alle EJBs gleich zu verwalten.

Die EJB Spezifikation in der Version 2.0 definiert drei unterschiedliche Typen von Enterprise Java Beans:

- *Entity Bean*: Eine Entity Bean repräsentiert eine persistente Entität, die in einer relationalen Datenbank gespeichert wird. Über Entity Beans ist es möglich, die persistenten Daten, die in einer Datenbank gespeichert vorliegen, abzufragen oder zu manipulieren. Es können generell zwei Typen von Entity Beans unterschieden werden.

  - *Container-managed Persistence (CMP)*: Bei diesem Entity Bean Typ kümmert sich der EJB Container um notwendige Datenbankkommunikationen, unter der Vorraussetzung, dass es sich bei der Datenbank um eine relationale Datenbank handelt. Die persistenten Attribute der Entity Bean müssen in einem Deployment Descriptor eingetragen werden.

  - *Bean-managed Persistence (BMP)*: Bei BMP Entity Beans werden Datenbankzugriffe von der Bean selbst ausgeführt. Die Bean EntwicklerIn ist somit für die Persistenz der Entity Bean verantwortlich. Ein Vorteil der BMP Entity Beans ist, dass Datenbankzugriffe optimiert durchgeführt werden können. Von Nachteil ist natürlich der erhöhte Programmieraufwand.

- *Session Bean*: Eine Session Bean ist ein transientes Objekt, welches als serverseitiger Agent Aktionen für einen Client ausführt. Es werden generell zwei Typen von Session Beans unterschieden.

  - *Stateful Session Bean*: Eine Stateful Session Bean beinhaltet Informationen über den Status der Kommunikation zwischen Client und Applikation. Dadurch kann eine Stateful Session Bean immer nur mit einem Client kommunizieren und nicht mit mehreren Clients gleichzeitig.

  - *Stateless Session Bean*: Eine Stateless Session Bean beinhaltet keine Statusinformationen zwischen verschiedenen Aufrufen, unterstützt aber mehrere Clients gleichzeitig. Somit kann durch den Einsatz von Stateless Session Beans eine bessere Skalierbarkeit der Applikationen erreicht werden.

- *Message-driven Bean*: Message-driven Beans sind in Bezug auf die Ausführung von Aktionen vergleichbar mit Session Beans. Der Unterschied der beiden Beantypen liegt darin, dass Message-driven Beans nur durch das Senden von Nachrichten über das Java Message Service (JMS) aufgerufen werden. Ein Client kann nicht, wie bei Session Beans üblich, über eine Schnittstelle auf eine Message-driven Bean zugreifen, sondern muss JMS als API für das Senden der Nachrichten verwenden. Message-driven Beans erlauben J2EE Applikationen asynchron zu kommunizieren.

Ein Client einer EJB kann vielartig sein, wie z.B. ein Servlet, ein Applet oder eine andere EJB. Dadurch kann eine komplexe Aufgabe auf mehrere Beans aufgeteilt werden und somit ist es möglich die Aufgabe in mehrere kleinere Teilaufgaben zu zerlegen.

Architektur der Terminverwaltung

Abbildung 5.3 gibt einen Überblick über die Architektur der Terminverwaltung. Die Applikation wird in drei Schichten eingeteilt: Präsentations-, Geschäftslogik- und Datenhaltungsschicht. Ein Client fordert seine Seiten über einen Controller an. Der Controller entscheidet, ob er sofort eine Antwort in Form einer Webseite zurücksendet oder vorher selbst eine Anfrage an den Applikationsserver stellen muss. Im Applikationsserver laufen die Geschäftslogikkomponenten ab. Diese kommunizieren mit einem Datenbankserver, um persistente Daten abzufragen, zu speichern oder zu ändern. Somit ist die durch das MVC-Muster geforderte Trennung erreicht. Die JSPs bzw. Servlets für die View- bzw. Controllerschicht befinden sich in einem Webserver. Die Geschäftslogik befindet sich als Session Beans gekapselt in einem Applikationsserver. Die Entity Beans für die persistenten Daten befinden sich auch im Applikationsserver und kommunizieren zusätzlich mit dem Datenbankserver, in dem die eigentliche persistente Speicherung vorgenommen wird.

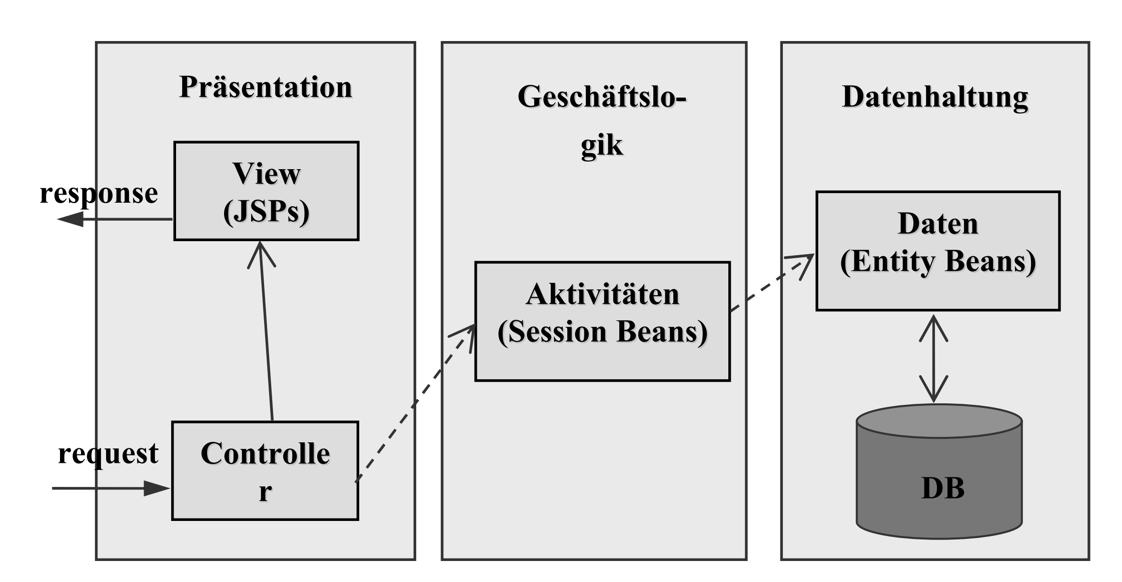


Abbildung 5.3: Architektur des Terminverwaltungssystems

Technologiewahl

Für die Fallstudie werden folgende Technologien eingesetzt: BEA Weblogic Server[1] in der Version 8.1 als J2EE Applikationsserver, Jakarta Tomcat[2] in der Version 4.0.1 als Webserver und PointBase als Datenbankserver. PointBase wird deshalb als relationale Datenbank verwendet, da diese mit dem BEA Weblogic Server kostenlos mitgeliefert wird.

MDA Werkzeugwahl

Für die Erstellung der Webapplikation wird das MDA-Werkzeug ArcStyler eingesetzt, da ArcStyler gerade für J2EE Anwendungen standardmäßig gute Unterstützung bietet. ArcStyler unterstützt das MVC-Muster und verfügt über ausgereifte Transformationen für die Generierung von J2EE Anwendungen aus UML-Modellen. Ein großer Vorteil des ArcStyler Werkzeuges ist die integrierte Testumgebung, die über spezielle Menüs für Applikations-, Datenbank- und Webserver konfiguriert werden kann. Durch die integrierte Testumgebung können Prototypen rasch entwickelt und getestet werden.

5.3 Durchführung des Projektes

In diesem Unterkapitel wird die Entwicklung der Terminverwaltung erläutert. Als MDA-Werkzeug wird ArcStyler eingesetzt, das ein markiertes PIM direkt zu Code transformiert und keine PSM-Modellebene unterstützt. Daher beschränkt sich das Vorgehensmodell des Entwicklungsprozesses auf Modellierung, Modell-zu-Code Transformation, Implementierung der restlichen Funktionalität und Verteilung bzw. Test.

Die zu erstellende Terminverwaltung wird in zwei Subsysteme unterteilt. Das erste Subsystem umfasst die Geschäftslogik und die persistenten Daten der Anwendung. Dieses Subsystem wird als eine EJB Applikation implementiert und im J2EE Applikationsserver bzw. Datenbankserver ablaufen. Im Folgenden wird dieses Subsystem als *EJB-Subsystem* bezeichnet. Das zweite Subsystem beinhaltet die Web Benutzeroberfläche und die Steuerung der Anwendung. Realisiert wird dieses Subsystem durch JSPs für die Benutzerschnittstelle und Java Servlets für die Steuerung. Die

---

[1] zu beziehen über http://www.bea.com (Stand: 1.12.2004)

[2] zu beziehen über http://jakarta.apache.org/tomcat/ (Stand: 1.12.2004)

Artefakte dieses Subsystems werden später im Webserver ablaufen. Im Folgenden wird dieses Subsystem als *Web-Subsystem* bezeichnet.

5.3.1 Modellierung

Als erster Schritt wird im ArcStyler ein neues Projekt angelegt. Danach müssen die benötigten MDA-Cartridges in das Werkzeug geladen werden. Für die Terminverwaltung werden insgesamt zwei MDA-Cartridges benötigt, nämlich eine für das EJB-Subsystem und eine weitere für das Web-Subsystem. Die Abbildung 5.4 zeigt die ausgewählten MDA-Cartridges für das Terminverwaltungsprojekt. Sobald die benötigten MDA-Cartridges ausgewählt wurden, stehen plattformspezifische Datentypen, Stereotype und Schlüsselwort/Wert-Paare als Modellelemente zur Verfügung.

Die MDA-Cartridge für das EJB-Subsystem sollte über Modellelemente für die Konzepte der J2EE Plattform verfügen und zusätzlich Unterstützung für den Applikationsserver BEA WebLogic Server bzw. für den Datenbankserver PointBase bieten. Diese Anforderungen erfüllt die mit ArcStyler standardmäßig mitgelieferte MDA-Cartridge BEA WebLogic Server 8.1, die kurz mit *WLS-Cartridge* bezeichnet wird.

Die MDA-Cartridge für das Web-Subsystem sollte Unterstützung für das MVC-Muster und für die Erstellung von Web-Benutzeroberflächen bieten. Zusätzlich wäre eine Unterstützung des Webservers Tomcat wünschenswert, um eine rasche Testmöglichkeit für Prototypen zu bieten. Mit dem ArcStyler Werkzeug wird standardmäßig eine *WebAccessors-Cartridge* mitgeliefert, welche den Anforderungen für die Erstellung des Web-Subsystems entspricht.

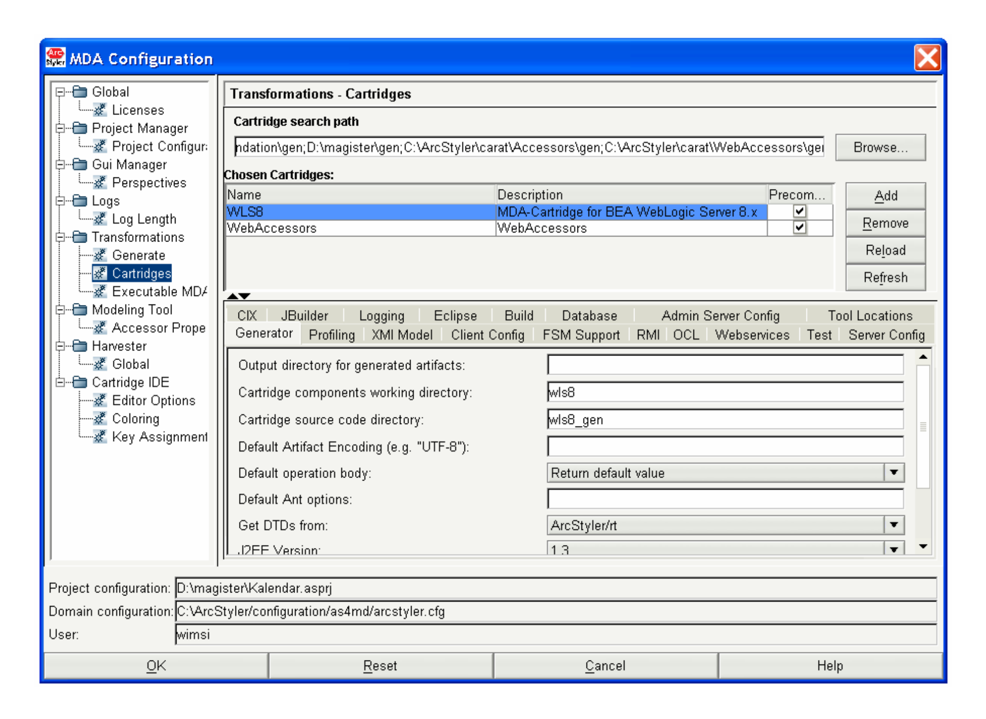


Abbildung 5.4: Konfiguration der MDA-Cartridges

Im Folgenden wird zuerst auf die Modellierung des logischen EJB-Subsystems eingegangen und danach die Modellierung des logischen Web-Subsystems erläutert. Sobald die beiden logischen Subsysteme modelliert sind, werden die physischen Komponenten der Anwendung modelliert.

Erstellung des logischen EJB-Subsystemmodells

In diesem Abschnitt wird auf die Modellierung der Geschäftslogik und der Datenhaltung eingegangen. Um die Modellierung des logischen EJB-Subsystems besser zu verstehen, werden zu Beginn dieses Abschnitts die grundlegenden Funktionalitäten der *WLS-Cartridge* aufgelistet. Die *WLS-Cartridge* bietet im J2EE-Kontext folgende Unterstützung für die EntwicklerInnen:

- Modellierung und Generierung der wichtigsten EJB 2.0 Elemente:
    - Session Beans (stateful, stateless)
    - Entity Beans (Container-managed Persistence, Bean-managed Persistence)
    - Message-driven Beans
    - Komponentenschnittstellen (*local*, *remote*)
- Modellierung und Generierung von physischen Paketen (*Java Archive*, *EJB Archive* und *Enterprise Application Archive*)
- Unterstützung für PointBase- und Oracle-Datenbanken

- Erstellung von Projektdateien für JBuilder und Eclipse
- ANT-basierte Unterstützung für die Erstellung, die Tests und die Verteilung der physischen Komponenten

Weiters werden von der WLS-Cartridge zwei spezielle Typen von UML-Diagrammen für die Modellierung von EJB-Applikationen angeboten, nämlich das *EJB-Klassendiagramm* und das *EJB-Komponentendiagramm*. Beide Diagrammarten sind mit zusätzlichen Funktionen in ihren Menüs ausgestattet, um häufig auftretende Arbeitsschritte zu erleichtern bzw. zu beschleunigen.
Für die persistente Datenhaltung müssen keine speziellen Diagramme, wie z.B. ER-Diagramme, erstellt werden, da die Datenbankspezifikationen direkt aus dem logischen Modell des EJB-Subsystems abgeleitet werden. In Kapitel 5.3.2 wird auf die automatische Generierung der Datenbankschemata aus dem Modell des EJB-Subsystems genauer eingegangen.

EJBs werden in einem EJB-Klassendiagramm als Klassen modelliert und mit dem Stereotyp *«ComponentSegment»* gekennzeichnet. In einem ComponentSegment werden alle EJB-Eigenschaften für Home-Schnittstelle, Remote-Schnittstelle, Implementierungsklasse und optionale Primärschlüsselklasse in einer UML-Klasse gekapselt. Die Abbildung 5.5 zeigt das EJB-Klassendiagramm für die beiden Entity Beans *Benutzer* und *Termin*.

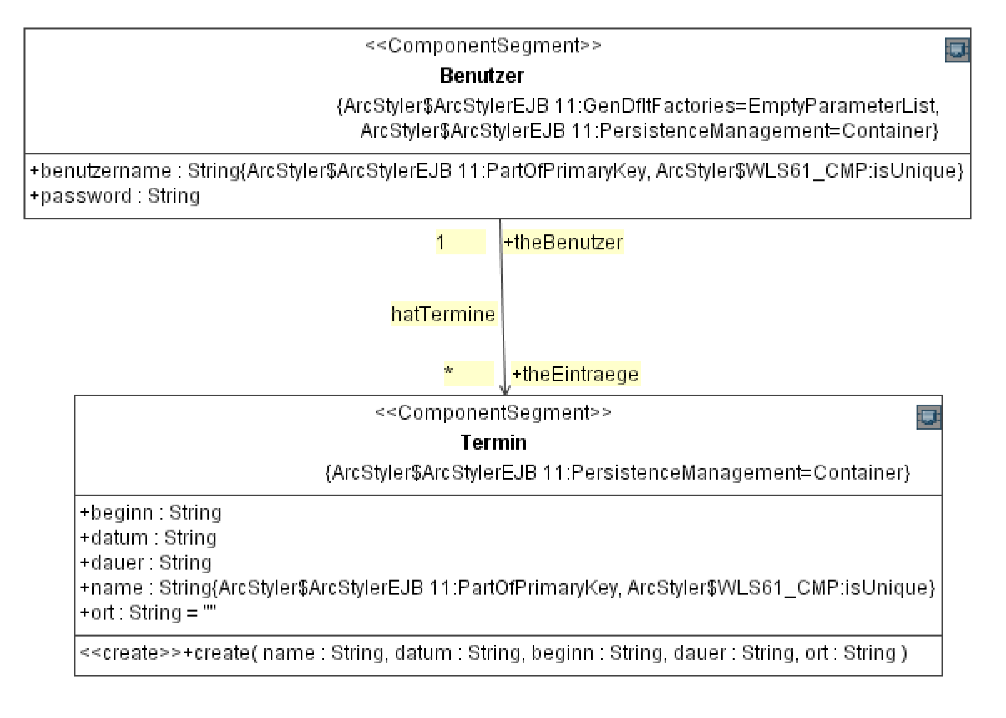


Abbildung 5.5: Auszug aus dem EJB-Klassendiagramm der Terminverwaltung

Die Eigenschaften einer Bean, wie z.B. der Typ der Bean (Entity, Session oder Message-driven), müssen durch Markierungen im Modell annotiert werden. Jedes ComponentSegment verfügt über ein MDA-Markierungsfenster. Über dieses Fenster ist die Karteikarte EJB 1.2/2.0 (Abbildung 5.6) verfügbar, in der die Realisierungsmöglichkeiten der Bean ausgewählt werden können.

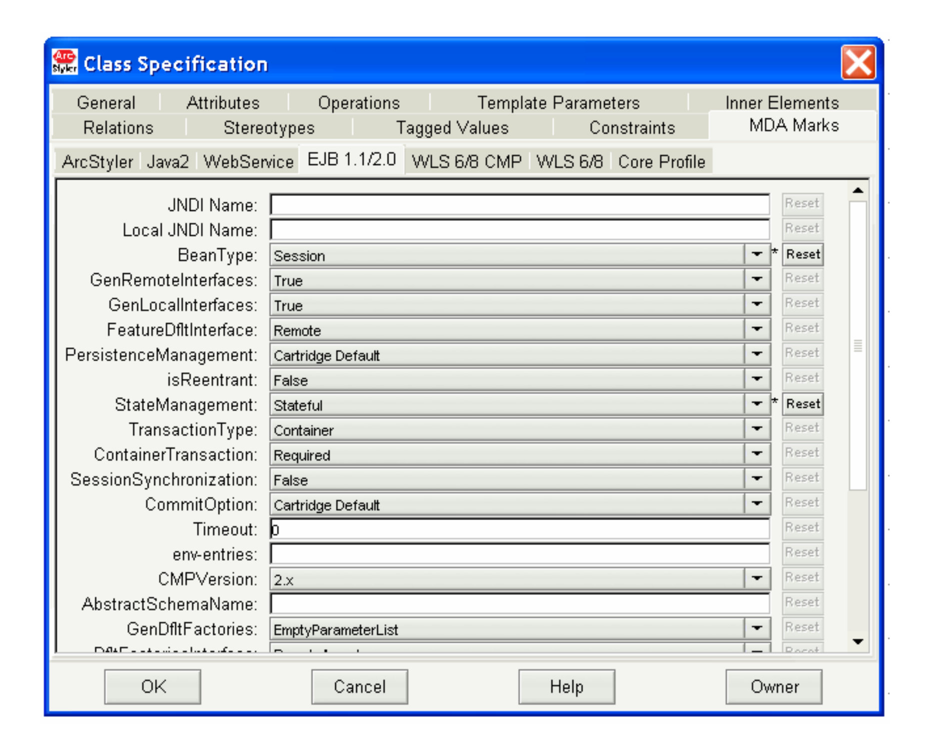


Abbildung 5.6: Markierungsfenster eines ComponentSegments

Erstellung des logischen Web-Subsystemmodells

Die *WebAccessors-Cartridge* fokussiert auf die View- und Controllerschicht des MVC-Musters und wird verwendet, um die Interaktion der Präsentationsschicht mit der darunterliegenden Geschäftslogikschicht zu modellieren. Die *WebAccessors-Cartridge* stellt zwei Hauptkonzepte zur Verfügung, nämlich *Representer-Klassen* für die Erstellung der Viewschicht und *Accessor-Klassen* zur Erstellung der Controllerschicht.

Um die Benutzerschnittstelle zu modellieren, werden Representer-Klassen eingesetzt. Diese werden als Klassen mit dem Stereotyp *«Representer»* notiert. Für jede Webseite ist eine eigene Representer-Klasse anzugeben, welche die Elemente der Benutzeroberfläche, wie z.B. Schaltflächen oder Textfelder, enthält. Eine Representer-Klasse kann somit als Sammlung von Elementen der Benutzerschnittstelle gesehen werden. Die Elemente der Benutzerschnittstelle werden der Representer-Klasse

über einen eigenen Menüpunkt zugewiesen. Über dieses Menü können einfache E-lemente, wie z.B. Checkboxen oder Textfelder, bis hin zu komplexeren Elemente, wie z.B. Frames oder Panels, der Representer-Klasse zugewiesen werden. Es sei aber darauf hingewiesen, dass die Spezifikation der Benutzerschnittstelle durch proprietäre Erweiterungen ermöglicht wird und sich nicht auf anerkannte Modellierungsstandards stützt. Somit sind diese Benutzerschnittstellendefinitionen nur im ArcStyler nutzbar.

Um die Steuerung der Benutzerschnittstelle modellieren zu können, werden im ArcStyler *Accessor-Klassen* angeboten. Diese werden als Klassen mit dem Stereotyp *«Accessor»* notiert. Eine Accessor-Klasse kann eine oder mehrere Representer-Klassen steuern bzw. anzeigen und weitere Accessor-Klassen benutzen. Eine Accessor-Klasse und ihre benutzten Representer-Klassen werden mit einer Abhängigkeitsbeziehung verbunden. Abbildung 5.7 zeigt das Klassendiagramm für die View- und Controllerschicht der Anmeldung eines Benutzers. Die Klasse *AnmeldenRep* enthält alle Informationen über die Benutzerschnittstelle und die Klasse *AnmeldenAcc* ist für die Steuerung der Anmeldung verantwortlich. Die Variablen in der View-Schicht müssen auch in der Controller-Schicht vorhanden sein, um die automatische Weitergabe der durch den Benutzer eingegebenen Werte in die Controller-Klassen zu gewährleisten, da diese Benutzereingaben erst in der Controller-Schicht verarbeitet bzw. überprüft werden. Dies wird durch *Resource Mappings* realisiert, die in Zustandsdiagrammen definiert werden können. Dabei wird eine Variable A einer Klasse A1 als Quelle und eine Variable B einer Klasse B1 als Ziel definiert (Vorraussetzung: A und B müssen kompatible Datentypen haben). Wird nun der Wert x der Variable A zugewiesen, wird der Wert x durch das definierte Resource Mapping automatisch der Variablen B zugewiesen. Dabei sei wieder darauf hingewiesen, dass es sich bei Resource Mappings um proprietäre Konzepte des ArcStyler Werkzeuges handelt.



Abbildung 5.7: Accessor-Diagramm für die Anmeldung eines Benutzers

Das dynamische Verhalten der Benutzerschnittstelle wird durch *Accessor-Aktivitätsdiagramme* beschrieben. Dabei wird jeder Accessor-Klasse ein Accessor-

Aktivitätsdiagramm zugewiesen. Accessor-Aktivitätsdiagramme sind mit proprietären Elementen angereicherte UML-Aktivitätsdiagramme und verfügen über Notationen für Zustände, obwohl sie explizit als Aktivitätsdiagramme bezeichnet werden. Die zwei wichtigsten Erweiterungskonzepte von Accessor-Aktivitätsdiagrammen stellen *Representer State* und *Embedded Accessor State* dar.

- *Representer State*: Ein Representer State ist ein Zustand, in dem eine zugewiesene Representer-Klasse angezeigt wird. Durch Representer States können auch Ereignisse die in der zugewiesenen Representer-Klasse auftreten, wie z.B. die Aktivierung einer Schaltfläche, berücksichtigt werden.

- *Embedded Accessor State*: Durch Embedded Accessor States wird es einer Accessor-Klasse ermöglicht, weitere Accessor-Klassen zu beinhalten bzw. zu benutzen. Dabei stellt ein Embedded Accessor eine Referenz auf eine Accessor-Klasse dar. Sobald in einem Aktivitätsdiagramm ein Embedded Accessor State erreicht ist, wird das Aktivitätsdiagramm der referenzierten Accessor-Klasse ausgeführt. Das Aktivitätsdiagramm der übergeordneten Accessor-Klasse wird nach dem Beenden des Aktivitätsdiagramms der referenzierten Accessor-Klasse fortgesetzt.

Abbildung 5.8 zeigt das Aktivitätsdiagramm für die Accessor-Klasse *Startseite*, welche die Hauptsteuerung der Terminverwaltung übernimmt. Der Representer State *Start* repräsentiert den oberen und unteren Frame der Benutzeroberfläche, die Representer-Klassen der eingebetteten Zustände werden im mittleren Frame angezeigt. Der Representer State *Start* enthält einen Representer States (*Ansicht*) und vier Embedded Accessor States (*Login*, *Logout, EintragHinzufuegen* und *EintragAendern*). Die Transitionen mit der Beschriftung |*SUB*|*::transitionsnamenl(final state)* werden aktiviert, sobald der Endzustand des Zustandsdiagrammes der eingebetteten Accessor-Klasse erreicht wurde. Die Transitionen mit der Beschriftung |*RE*|*::transitionsnamen* schalten, sobald eine entsprechende Schaltfläche der Representer-Klassen aktiviert wurde.

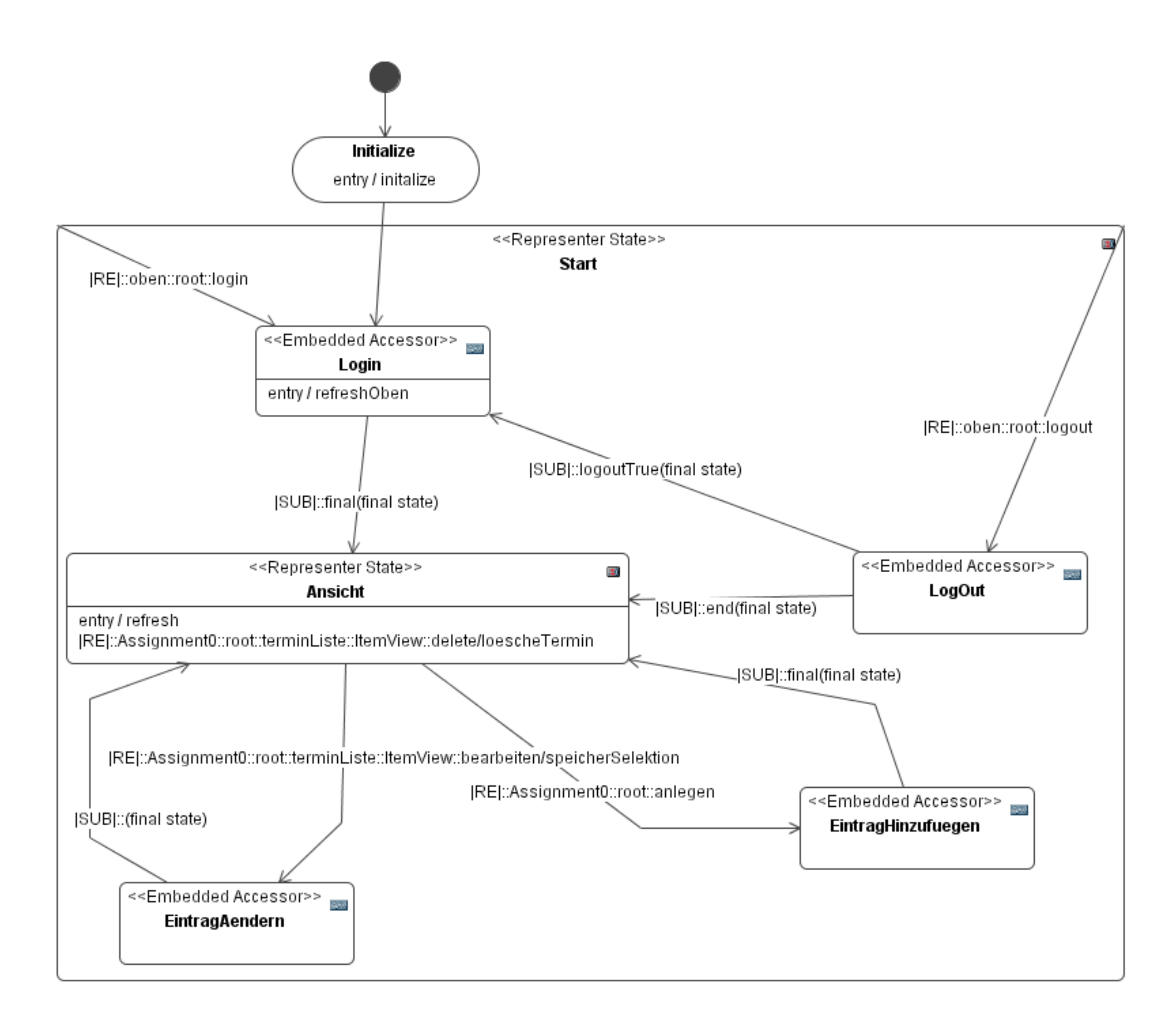


Abbildung 5.8: Aktivitätsdiagramm für die Accessor-Klasse Hauptseite

Modellierung der physischen Komponenten

Da die logische Struktur der Terminverwaltung besprochen wurde, wird nun mit der Modellierung der physischen Verteilung der Komponenten fortgefahren. Für das EJB-Subsystem wird ein EJB-Archiv names *ejbs*, welches die modellierten Elemente des EJB-Subsystems enthält, erstellt. Das EJB-Archiv wird später im EJB Container eingesetzt. Das EJB-Archiv wird in einem Komponentendiagramm als Komponente mit dem Stereotyp *«EJBArchive»* modelliert. Für das Web-Subsystem wird ein Webapplikationsarchiv benötigt, das später im Webserver eingesetzt wird. Dieses Webapplikationsarchiv wird als Komponente mit dem Namen *webapp* und mit dem Stereotyp *«Webapplication»* modelliert. Der Stereotyp *«Webapplication»* verlangt die Angabe eines *RootAccessors*, der die Startseite der Webapplikation repräsentiert. Da das Webapplikationsarchiv das EJB-Archiv verwendet, muss eine Abhängigkeit vom Webapplikationsarchiv zum EJB-Archiv eingezeichnet werden. Abbildung 5.9 zeigt das Komponentendiagramm für die Terminverwaltungsapplikation.

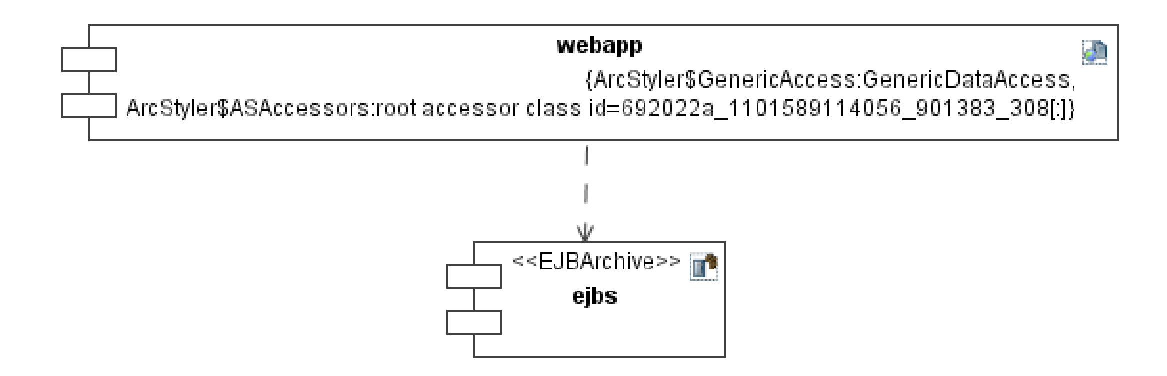


Abbildung 5.9: Komponenten der Terminverwaltung

5.3.2 Generierung der Artefakte

Dieses Unterkapitel beschäftigt sich mit der Codegenerierung. In diesem Schritt der Softwareentwicklung werden der Codegenerator und die MDA-Cartridges aktiviert, um die notwendigen Artefakte für ein lauffähiges Softwaresystem aus UML-Modellen zu erstellen. Die Erklärung der automatischen Codegenerierung wird für die beiden Subsysteme EJB-Subsystem und Web-Subsystem getrennt durchgeführt. Dabei wird für beide Subsysteme folgende Vorgehensweise für den Transformationsprozess angewendet: Konfiguration des Codegenerators, Modellverifikation und Aktivierung der Transformation.

Transformation des EJB-Subsystems

Bevor der Codegenerierungsprozess gestartet werden kann, muss der Codegenerator konfiguriert werden. Im Besonderen müssen folgende Einstellungen über das MDA Konfigurationsmenü vorgenommen werden:

- Ausgabepfad der zu generierenden Artefakte
- Konfiguration des Datenbankservers
- Konfiguration des Applikationsservers

Um sicherzustellen, dass nur fehlerfreie Modelle transformiert werden, ist die Überprüfung der Struktur des Modells notwendig. ArcStyler bietet durch seine WLS-Cartridge eine eigene *Verify* Funktion für EJB-Modelle. Bei der Ausführung der Modellverifikation werden Warnungen/Fehlermeldungen für nicht optimale/fehlerhafte Modellierungen ausgegeben.

Der Transformationsprozess wird mit der Auswahl des EJB-Subsystems im Modellbrowser und der Aktivierung des Befehls *Generate* ausgeführt. Im Log-Fenster des ArcStylers werden alle generierten Artefakte mit ihren Systempfaden mitprotokolliert. ArcStyler leitet aus einem ComponentSegment folgende Artefakte ab:

- Remote Interface (optional)
- Local Interface (optional)
- Remote Home Interface (optional)
- Local Home Interface (optional)
- Bean Implementierungsklasse (notwendig)
- Deployment Descriptor (notwendig)
- Primärschlüsselklassen (werden nur für Container-managed Persistence Entity Beans generiert)

Als Beispiel für die generierten Artefakte des EJB-Subsystems wird folgendes Codesegment aus der Datei AnmeldungBean angegeben. Für Bean Implementierungsklassen werden einerseits vorgeschriebene Methoden für den EJB Container (wie z.B. die Methode *ejbActivate()*) und andererseits Methodenrümpfe mit geschützten Bereichen für selbstdefinierte Methoden (wie z.B. die Methode *authentifizieren()*) durch die WLS-Cartridge erzeugt.

```
package ejbCalendarium.anmelden;

...

public  class AnmeldungBean implements javax.ejb.SessionBean{

  public void ejbCreate() throws CreateException{

  }

  public void ejbActivate(){

  }
  ...

  public boolean authentifizieren( java.lang.String
     benutzer,java.lang.String password){
     /* START OF PROTECTED AREA  <<authentifizieren
          :String:String>> */

     // insert custom code here

     /* END OF PROTECTED AREA 1604e74e0000006e(C) */
  }
  ...
}
```

Die persistenten Attribute und Assoziationen einer Entity Bean werden im Allgemeinen in relationalen Datenbanken gespeichert. Dafür wird ein Datenbankschema benötigt, das durch den ArcStyler automatisch und vollständig vom EJB-Modell abgeleitet werden kann. Für jede Entity Bean wird eine eigene Tabelle erstellt. Der Name der Entity Bean wird standardmäßig als Tabellenname übernommen, falls kein anderer Name im MDA-Eigenschaftenfenster der Entity Bean spezifiziert wurde. Für jedes Attribut bzw. jede Rolle wird eine eigene Spalte in der Tabelle angelegt. Der Name des Attributes oder der Rolle wird dabei als Spaltenname verwendet. Es kann aber über das MDA-Eigenschaftenfenster ein beliebiger Name für die Spalte vergeben werden. Die WLS-Cartridge verwendet spezielle Abbildungen für die Datentypen. Für eine konkrete Auflistung der Abbildungen zwischen Java-Datentypen und SQL-Datentypen sei auf [IO-03d] verwiesen. Der folgende SQL-Code für das Anlegen und Löschen der Tabellen wird automatisch aus den beiden Entity Beans Termin und Benutzer generiert.

```
/*Skript zum Anlegen der Tabellen*/
CREATE TABLE Eintrag (
  name VARCHAR(300) NOT NULL ,
  ort VARCHAR(2000) ,
  datum VARCHAR(2000) ,
  beginn VARCHAR(2000) ,
  dauer VARCHAR(2000) ,
  theBenutzer VARCHAR(300),
  PRIMARY KEY (name)
);
CREATE TABLE Benutzer (
  benutzername VARCHAR(300) NOT NULL ,
  password VARCHAR(2000) ,
  PRIMARY KEY (benutzername)
);


/*Skript zum Löschen der Tabellen*/
DROP TABLE Eintrag;
DROP TABLE Benutzer;
```

Transformation des Web-Subsystems

Bevor die Transformation des Web-Subsystems zu Code begonnen werden kann, muss wie bei der Transformation des EJB-Subsystems der Codegenerator konfiguriert werden. Für die WebAccessors-Cartridge reicht die Konfiguration der Ausgabepfade für die zu generierenden Artefakte und die Konfiguration des Tomcat Webservers. Um die Modelle auf Konsistenz zu prüfen ist die WebAccessors-Cartridge ebenso wie die WLS-Cartridge mit einer *Verify* Funktion ausgestattet. Diese Möglichkeit sollte vor jeder Modelltransformation genutzt werden, um keine fehlerhaften

Modelle zu transformieren. Mittels der Funktion *Generate* wird die Transformation des Web-Subsystemmodells gestartet.

Eine Accessor-Klasse wird in eine Java-Klasse mit dem Namen der Accessor-Klasse transformiert. Das zur Accessor-Klasse gehörende Aktivitätsdiagramm wird als innere Klasse mit dem Namen *_vertex_top* implementiert. Die innere Klasse *_vertex_top* beinhaltet für jeden Zustand oder Aktivität des Aktivitätsdiagrammes eine innere Klasse mit dem Namen *_vertex_x* wobei x für den Namen des jeweiligen Zustandes oder der jeweiligen Aktivität steht.

Das folgende Codefragment zeigt einen kleinen Auszug der generierten Datei AnmeldeAccessor. Jeder Zustand wird als innere Klasse der inneren Klasse *_vertex_top* implementiert. Die inneren Klassen erben abhängig vom Typ der Zustände. Für jeden Zustandstyp gibt es eine äquivalente Java-Klasse, die besondere Funktionalität für diesen Zustandstyp bietet. Die Klasse *_vertex_Anmeldung* erbt zum Beispiel von der Klasse *com.io_software.catools.jspacc.accessor.fsm.RepresenterState*, da der Zustand Anmeldung von Typ *RepresenterState* ist. Die inneren Klassen sind mit speziellen Methoden, die sie von ihren Elternklassen erben, ausgestattet.

```
package webAccessors.anmelden;

public class CalendarAccessor extends
  com.io_software.catools.jspacc.accessor.AccessorGenBaseImpl{

  public CalendarAccessor(java.lang.String benutzername,
         java.lang.String password, java.lang.String message,
         ejbCalendarium.Kalendar kalendar){

  }

  public class _vertex_top extends
     com.io_software.catools.picasso.fsmrt.TopState{

       public class _vertex_start extends
         com.io_software.catools.picasso.fsmrt.InitialState{

       }
       public class _vertex_Anmeldung extends
         com.io_software.catools.jspacc.accessor.fsm.
         RepresenterState{

       }

       public class _vertex_check extends
         com.io_software.catools.picasso.fsmrt.State
       {

       }
   }
}
```

Eine Representer-Klasse wird in eine Java-Klasse mit dem Namen der Representer-Klasse und in eine Java Server Page mit dem Namen der Representer-Klasse transformiert. Die Benutzerschnittstellenelemente einer Representer-Klasse werden als innere Klassen in der Java-Klasse und als Methodenaufrufe für die Erzeugung der Elemente in der Java Server Page berücksichtigt.

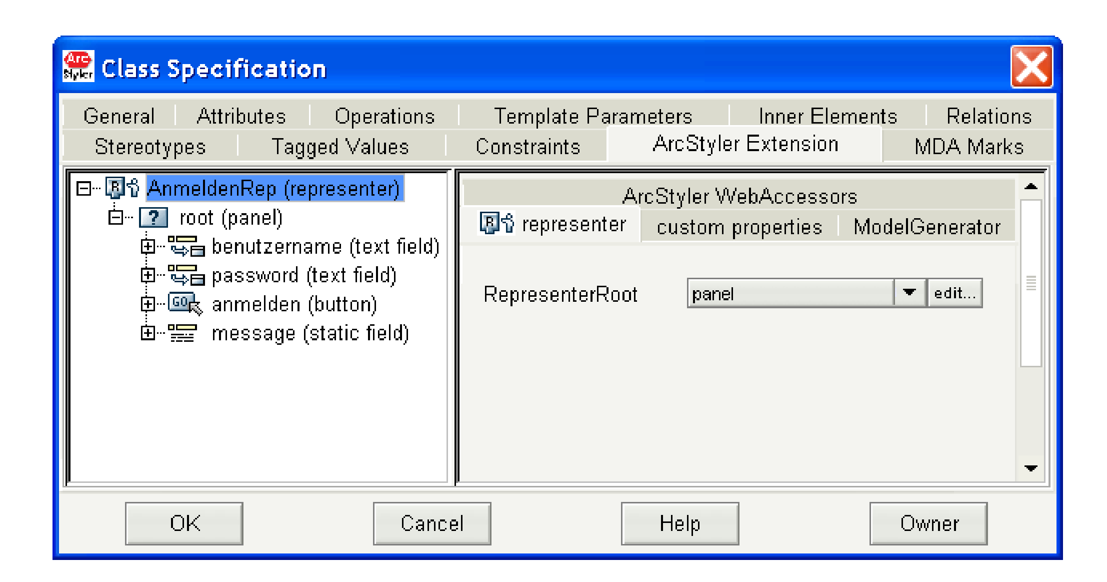


Abbildung 5.10: Benutzerschnittstellenelemente der Klasse AnmeldeView

Die folgenden zwei Codefragmente werden automatisch aus der Representer-Klasse *AnmeldenView* erzeugt. Die Benutzerschnittstellenelemente der Klasse AnmeldeView werden in Abbildung 5.10 aufgelistet. Das erste Codefragment repräsentiert einen Auszug aus der AnmeldeView Java-Klasse. Die Benutzerschnittstellenelemente der Representer-Klasse werden als innere Klassen implementiert, wobei die innere Klasse für das Panel *root* alle weiteren inneren Klassen für die Elemente beinhaltet.

```
package webAccessors.anmelden;


public class AnmeldeView extends
     com.io_software.catools.jspacc.bob.JspActionContainer
{

  public AnmeldeView(java.lang.String benutzername,
         java.lang.String password, java.lang.String message)
  {
     ...
  }


  public class _ce_Root extends ASPanelModel
  {

    public class _ce_Benutzername extends ASTextFieldModel
```

```
        {

          public _ce_Benutzername(String p_ElementName)
          {
            super(p_ElementName);
            setMultiline(false);
          }

        }

        public class _ce_Password extends ASTextFieldModel
        {
             ...
        }

        public class _ce_Anmelden extends ASButtonModel
        {
             ...
        }

        public class _ce_Message extends ASStaticFieldModel
        {
             ...
        }
      }
    }
```

Das zweite Codefragment repräsentiert einen Auszug der AnmeldeView Java Server Page. In dieser Java Server Page werden die Benuterschnittstellenelemente durch Methodenaufrufe aus dem Objekt *me* vom Typ *AnmeldeView* erzeugt. Zusätzlich werden Java-Script-Methoden eingebunden, um die Elemente zu formatieren.

```
webAccessors.anmelden.AnmeldeView me = (webAccessors
     .anmelden.AnmeldeView)request.getAttribute("instance");

<html>
<head>
 <title><%=me.getTitle()%></title>
</head>


<%-- start of element 'root' --%>
<%  webAccessors.anmelden.AnmeldeView._ce_Root root__model =
      me.getRoot();
    currentModel=root__model;
%>

 <%-- start of element 'benutzername' --%>
 <%
   webAccessors.anmelden.AnmeldeView._ce_Root._ce_Benutzername
      root_benutzername__model = root__model.getBenutzername();
   currentModel=root_benutzername__model;
 %>
 <%@ include file="/&resources&/jspinc/ASLabelForElement.
      jspinc" %>
 <%@ include file="/&resources&/jspinc/ASTextField.jspinc" %>

 <%-- start of element 'password' --%>
 <%
   webAccessors.anmelden.AnmeldeView._ce_Root._ce_Password
```

```
    root_password__model = root__model.getPassword();
  currentModel=root_password__model;
%>
<%@ include file="/&resources&/jspinc/ASLabelForElement.
     jspinc" %>
<%@ include file="/&resources&/jspinc/ASTextField.jspinc" %>

<%-- start of element 'anmelden' --%>
<%
  webAccessors.anmelden.AnmeldeView._ce_Root._ce_Anmelden
    root_anmelden__model = root__model.getAnmelden();
  currentModel=root_anmelden__model;
%>
<%@ include file="/&resources&/jspinc/ASButton.jspinc" %>

<%-- start of element 'message' --%>
<%
  webAccessors.anmelden.AnmeldeView._ce_Root._ce_Message
    root_message__model = root__model.getMessage();
  currentModel=root_message__model;
%>
<%@ include file="/&resources&/jspinc/ASLabelForElement.
     jspinc" %>
<%@ include file="/&resources&/jspinc/ASStaticField.jspinc" %>
<%-- end of element 'root' --%>

</body>
</html>
```

5.3.3 Ergänzung des generierten Codes

Dieses Unterkapitel beschäftigt sich mit der manuellen Codeergänzung der generierten Artefakte. ArcStyler bietet nicht für jeden Bereich 100 % Codegenerierung und somit müssen einige Codeteile von den ProgrammiererInnen händisch erstellt werden. Für das Arbeiten auf Codeebene empfiehlt sich die Nutzung der automatischen Erstellungsfunktion von Projektdateien für Eclipse oder JBuilder, da eine Programmier-IDE eine große Hilfe für die manuelle Codeergänzung darstellt. Damit die manuellen Ergänzungen im Code bei der nächsten Modell-zu-Code Transformation nicht verloren gehen, bietet ArcStyler sogannte geschützte Bereiche im generierten Code. Diese geschützten Bereiche werden durch den Codegenerator nicht überschrieben.

Im EJB-Subsystem stellen die ersten Codeergänzungsschritte die Vervollständigung der benutzerdefinierten *create*-Methoden von Entity Beans dar. Da in den meisten Fällen in den create-Methoden nur die übergebenen Parameter den Attributen der Entity Bean mittels *Set*-Methodenaufrufen zugewiesen werden, sind diese Ergänzungen rasch und ohne intellektuellen Aufwand durchzuführen.

Die Implementierung der Geschäftslogikmethoden von den Session Beans ist hingegen um einiges aufwändiger, denn für die Geschäftslogikmethoden werden nur die

Methodenrümpfe automatisch generiert, die restliche Implementierung bleibt der ProgrammierIn überlassen.
Als Beispiel für die Implementierung von Geschäftslogikmethoden wird die Methode *authentifizieren* der AnmeldeBean angeführt. Die Methode *authentifizieren* hat die Aufgabe, die Überprüfung von Benutzername/Passwortkombinationen vorzunehmen. Der dafür verantwortliche Code wird in den geschützten Bereich der Methode *authentifizieren* von Hand eingetragen. Dieses Codebeispiel verdeutlicht, dass bei der Implementierung der Geschäftslogik sehr wohl Kenntnisse über den generierten Code und der Softwareplattform gefordert sind, wie z.B. Wissen über die Eigenschaften der Methode *getEJBRef_EjbCalendarium_anmelden_Benutzer()*.

```
public boolean authentifizieren( java.lang.String
               benutzer,java.lang.String password){
  /* START OF PROTECTED AREA  <<authentifizieren
       :String:String>> */
   // insert custom code here

   BenutzerHome benutzerHome = this.getEJBRef_EjbCalendarium
                              _anmelden_Benutzer();
   Benutzer theBenutzer = null;

   try{
        theBenutzer = (Benutzer) benutzerHome.findByPrimaryKey
                               (benutzer);
   }catch(Exception e){

   }

   try{
        if (theBenutzer != null && password.equals(
            theBenutzer.getPassword())){
            return true;   //Benutzername/Passwortkombination
                           //korrekt
        }
   }catch(java.rmi.RemoteException re){

   }

   return false;
  /* END OF PROTECTED AREA 1604e74e0000006e(C) */
}
```

In den Controller-Klassen des Web-Subsystems sind weitere Codeergänzungen notwendig. Das Verhalten der Controller-Klassen wird durch ihre Accessor-Aktivitätsdiagramme definiert. In den Accessor-Aktivitätsdiagrammen sind nur die Deklarationen der Methoden für Zustände und Zustandsübergänge definierbar, nicht aber die konkreten Aktionen der Methoden. Diese Aktionen können erst auf Codeebene spezifiziert werden, da ArcStyler keine Definitionsmöglichkeiten von Aktionen auf Modellebene anbietet.

Als Beispiel für die Codeergänzungsarbeiten im Controller Bereich soll die Implementierung der Methode *checkLogin* der Klasse *AnmeldeAccessor* dienen. Die Methode *checkLogin* überprüft durch die Verwendung der Anmelde Session Bean den eingegebenen Benutzernamen und das eingegebene Passwort auf ihre Gültigkeit. Für diese Methode ist es notwendig, die benutzerdefinierten Signale des Accessor-Aktivitätsdiagramms der Controller-Klasse durch spezielle Methodenaufrufe versenden zu können, um den im Accessor-Aktivitätsdiagramm modellierten Ablauf auf Codeebene zu realisieren. Falls die Benutzername/Passwort-Kombination gültig ist, wird ein *ok*-Signal versendet, um in die Benutzerhauptansicht zu gelangen. Falls aber die Benutzername/Passwort-Kombination ungültig ist, wird eine entsprechende Fehlermeldung auf der Anmeldeseite angezeigt.

```
transition.addAction(new com.io_software.catools.picasso.
                      fsmrt.Action("authorizieren"){

     public void perform(com.io_software.catools.picasso.
                          fsmrt.Event a_Event){
        /* START OF PROTECTED AREA
          <<692022a_1100984575411_584415_109>> */
          // insert custom code here

          try{
               benutzername = getAssignment0Representer().
                              getRoot().getBenutzername().
                              getFormattedValue();
               password = getAssignment0Representer().getRoot()
                          .getPassword().getFormattedValue();
               try{
                    CalendarAccessor ca = CalendarAccessor.this;
                    ejbCalendarium.anmelden.Anmeldung al =
                         ca.getKalendar().getSignOn();

                    if (al.authentifizieren(ca.getBenutzername()
                        , ca.getPassword())){
                        ca.getKalendar().setLoggedUserById(
                                         ca.getBenutzername());
                         postEvent("ok"); //Benutzer berechtigt
                    }else{
                         message = "Login failed";
                    }
               }
               catch(Exception e){
                    System.out.println(e.toString());
                    message = "Benutzername existiert nicht";
               }
               initAssignment0RepresenterData(); //anzeigen der
                                                 //Fehlermeldung
          }
          catch(Exception ee){
               System.out.println(ee.toString());
          }
       /* END OF PROTECTED AREA 7cf6600500000034(C) */
     }
});
```

Die Gestaltung der Benutzerschnittstelle erfordert weitere Arbeiten auf Codeebene, da im ArcStyler nur die Elemente der Benutzerschnittstelle definiert werden können, nicht aber ihre Anordnung und ihre Darstellungsmöglichkeiten. Somit müssen Nacharbeiten in den JSP-Dateien durchgeführt werden, um das gewünschte Layout zu erhalten. Auf die Layoutarbeiten wird aber weiters nicht näher eingegangen, da diese in jedem gewöhnlichen HTML-Editor zu bewerkstelligen sind.

Da ArcStyler über eine eigene MDA-Cartridge für den BEA WebLogic Applikationsserver verfügt und diese für die Entwicklung der Terminverwaltung eingesetzt wird, brauchen keine Anpassungen für die generierten Deployment Descriptors und SQL-Skripte vorgenommen werden. Die für den BEA WebLogic Applikationsserver spezifischen Einstellungen werden bereits durch speziell angepasste Transformationen der WLS-Cartridge berücksichtigt.

5.3.4 Verteilung und Test

Dieser Abschnitt beschäftigt sich mit der Verteilung und dem Test der Terminverwaltung. Hierfür bietet ArcStyler besonders gute Unterstützung für die EntwicklerInnen. Neben den Artefakten für die Applikation werden durch die WLS-Cartridge zusätzliche ANT-Skripte generiert, welche die Erstellung und die Verteilung der einsatzfähigen Archive (z.B. JAR, WAR oder EAR) vornehmen. ArcStyler verfügt über ein eigenes ANT-Fenster (Abbildung 5.11), in dem alle verfügbaren ANT-Befehle zu einer markierten Komponente angezeigt werden und durch den Benutzer gestartet werden können, ohne dafür in die Kommandozeile wechseln zu müssen.

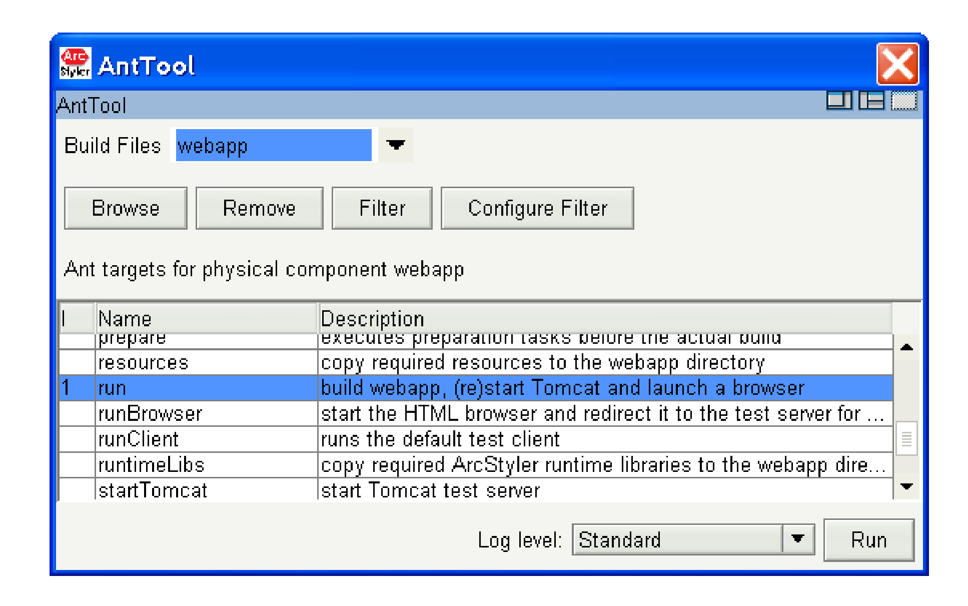


Abbildung 5.11: ANT-Fenster des ArcStylers

Verteilung und Test der EJB-Applikation

Es stehen ANT-Skripte für das Starten der Datenbank und das Anlegen bzw. Löschen der Tabellen für die Terminverwaltung zur Verfügung. Auch für das Kompilieren der EJBs und für das Packen der kompilierten EJBs zu einem Enterprise Archive werden Skripten angeboten. Sogar für das Starten des Applikationsservers und für die Verteilung des Enterprise Archives werden ANT-Skripten bereitgestellt. Schließlich kann ein Testclient mittels ANT-Skript gestartet werden, um die Funktionalität der EJB-Applikation zu testen.

Verteilung und Test der Webapplikation

Mit dem *run* ANT-Befehl werden alle Java-Klassen und JSPs kompiliert. Weiters wird eine WAR-Datei erstellt, der Tomcatserver gestartet und die Webapplikation im Tomcatserver verteilt. Danach wird der Standardbrowser gestartet, der die Startseite der Webapplikation anzeigt. Somit kann auch die Webapplikation vom ArcStyler aus rasch verteilt und getestet werden.

5.4 Resümee der Fallstudie

Dieses Unterkapitel beschäftigt sich mit den Erfahrungen der Fallstudie Terminverwaltung. Dabei wird speziell auf die Unterstützung des MDA-Werkzeugs ArcStyler für die Entwicklung der Terminverwaltung eingegangen.

Das MDA-Werkzeug ArcStyler generierte zwar den Großteil des Codes der Terminverwaltungsanwendung, doch die Implementierung der Geschäftslogikmethoden musste von Hand durchgeführt werden. Das Hinzufügen der Geschäftslogik stellte sich als äußerst aufwendig heraus und setzte Kenntnisse über den generierten Code voraus. Als ein Hauptproblem stellte sich die fehlende Spezifikationsmöglichkeit von Aktionen auf Modellebene heraus. Deshalb konnten die Methodeninhalte erst auf Codeebene und nicht schon auf Modellebene definiert werden.
ArcStyler bietet zwar die Möglichkeit, aus den erstellten Modellen Code für die .NET Plattform zu generieren, doch die Methoden müssen in C# neu implementiert werden. Generell kann gesagt werden, dass Funktionalität, die erst auf Codeebene und nicht schon auf Modellebene definiert wird, für Transformationen des Modells für Implementierungen, welche auf anderen Plattformen basieren, nicht genutzt werden kann und für diese Implementierungen neu erstellt werden muss. Diese Problematik zeigt einmal mehr, dass noch viele Verbesserungen in den MDA-Werkzeugen notwendig sind, da noch immer viele Arbeitsschritte erst auf Codeebene durchgeführt werden können. Zusammenfassend können folgende positive und negative Aspekte der Erstellung der Terminverwaltung durch das MDA-Werkzeug ArcStyler angeben werden:

Positive Aspekte

- Der Infrastrukturcode für EJBs, wie z.B. Schnittstellen oder Deployment Descriptors, wird vollständig automatisch generiert.

- Das Datenbankschema für die Speicherung der persistenten Attribute und Assoziationsrollen der Entity Beans wird vollständig aus dem EJB-Modell generiert.

- Die WebAccessors-Cartridge ermöglicht eine gute Trennung zwischen der View- und der Controllerschicht.

- Da ArcStyler über eine integrierte Testumgebung verfügt, können Prototypen rasch erstellt und getestet werden, ohne die ArcStyler Umgebung zu verlassen. Die automatisch generierten Benutzeroberflächen sind zwar nicht formatiert, stellen aber trotzdem eine gute Black Box-Testmöglichkeit dar.

- Durch die Generierung des Infrastrukturcodes wird ein fester Coderahmen für alle Dateien vorgegeben. Dadurch kann die Codequalität gesteigert und die Wartung verbessert werden.

- Durch den Einsatz des MDA-Paradigmas verfügt man durch das PIM über eine gute Dokumentation, die nicht vom Code abweicht, da dieser vom Modell abgeleitet wurde.

Negative Aspekte

- Es werden keine Aktivitäts- oder Zustandsdiagramme für Geschäftslogikmethoden geboten. Somit müssen Geschäftslogikmethoden bis auf die Methodensignatur manuell erstellt werden.

- Einige Methoden in den Controllerklassen müssen mit manuell erstelltem Code ergänzt werden.

- Da die händische Programmierung nicht gänzlich erspart bleibt, werden tiefgehende Kenntnisse über den generierten Code benötigt, um die aus dem Modell nicht generierbare Funktionalität zu implementieren.

- Bei der Verschiebung oder Umbenennung von Paketen oder Klassen in der Elementhierarchie werden bei der nächsten Transformation neue Quelltextdateien generiert. Sind in den alten Quelltextdateien manuelle Änderungen vorgenommen worden, werden diese in die neuen Dateien nicht automatisch übernommen. Die EntwicklerIn muss die manuellen Ergänzungen von den alten in die neuen Dateien händisch übertragen.

- Über verfügbare Methoden der Benutzeroberflächenelemente der WebAccessors-Cartridge werden keine Dokumentationen angeboten. Wird zum Beispiel ein Textfeld als Passwortfeld benötigt, so ist ein mühsames Suchen im Quelltext der mitgelieferten Beispielanwendungen notwendig.

- Für ein effektives Arbeiten mit dem ArcStyler Tool muss eine lange Einarbeitungszeit in Kauf genommen werden, denn EntwicklerInnen müssen eine neue Form der Programmierung erlernen.

- ArcStyler verwendet proprietäre Metamodelle für die WLS-Cartridge und die WebAccessors-Cartridge. Dieses Faktum macht die Nutzung der erstellten Modelle in anderen MDA-Werkzeugen unmöglich.

- Der generierte Code ist anfangs schwer verständlich. In den Klassen der Controller- und Viewschicht werden viele innere Klassen definiert und der Aufbau der JSPs ist anfangs auch schwer zu durchschauen. Weiters sei noch auf die ineffizienten SQL-Skripten hingewiesen. Beispielsweise werden die Felder für Attribute vom Datentyp String (wie z.B. das Passwort eines Benutzers) standardmäßig als VARCHAR(2000) angelegt. Diese Felder können und sollten im ArcStyler über die Markierungsfenster genauer spezifiziert werden.

- Treten beim Testen der Applikation Fehler auf, ist es für die EntwicklerIn sehr schwer zu entscheiden, ob man diese auf Modellebene beheben soll bzw. kann oder ob diese Fehler nur auf Codeebene beseitigt werden können. In diesem Zusammenhang wäre es weiter interessant zu klären, inwieweit ArcStyler manuelle Codeergänzungen ins Modell aufnehmen kann bzw. ob und in welchem Umfang ein Reverse Engineering (d.h. Code-zu-Modell Transformation) möglich ist.

Die Fallstudie zeigte, dass eine Verwendung des ArcStylers nur dann Sinn macht, wenn mehrere Projekte mit diesem Werkzeug und ähnlichen Plattformen folgen sollen. Wegen der langen Einarbeitungszeit und der möglicherweise selbst zu erstellenden Transformationen ist von einem einmaligen Einsatz eines MDA-Werkzeuges abzuraten.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Model Driven Architecture verspricht einen offenen und herstellerneutralen Ansatz für die Generierung von lauffähigen Anwendungen aus plattformunabhängigen Modellen. Da der Fokus der Entwicklung auf der Geschäftsebene liegt, sollen Softwaresysteme ohne Wissen über die einzusetzende Softwareplattform entwickelt werden. In den letzten vier Jahren wurden viele Forschungsaktivitäten im MDA Bereich unternommen und einige MDA-Werkzeuge konnten sich bereits am Markt behaupten.

Die Evaluierung der MDA-Werkzeuge und die Fallstudie zeigten, dass zurzeit nur wenige Vorteile des theoretischen MDA-Ansatzes in der Softwareentwicklung praktisch genutzt werden. Im Allgemeinen kann nur der Infrastrukturcode automatisch generiert werden, die lauffähige Anwendung jedoch nicht. Die Methodenrümpfe der Geschäftslogikmethoden müssen großteils von Hand ausprogrammiert werden. Von einem anderen Blickwinkel betrachtet entlastet die automatische Generierung des Infrastrukturcodes die EntwicklerInnen und können sich somit mehr auf die Implementierung der Geschäftslogik konzentrieren. Es können weiters Verbesserungen der Entwicklungszeit und der Codequalität erreicht werden. Leider wird durch diesen Ansatz noch nicht das gesamte Potential der MDA genutzt.

Einen weiteren Problembereich stellen die derzeitigen Transformationen dar. Solange es seitens der OMG keinen einheitlichen Standard für eine Transformationssprache gibt, werden Transformationen durch proprietäre Sprachen der aktuellen MDA-

Werkzeuge definiert. Somit ist eine Transformation nur für ein bestimmtes MDA-Werkzeug nutzbar und die Entstehung eines Marktes für Transformationen nur begrenzt realisierbar.

Auch das Versprechen von MDA, dass die EntwicklerInnen keine Kenntnisse über die eingesetzten Softwareplattformen benötigen, wird zu diesem Zeitpunkt nicht erfüllt. Die EntwicklerInnen benötigen sehr wohl Kenntnisse über die Softwareplattformen, da sie die PIMs mit plattformspezifischen Details markieren müssen. Diese Problematik wird durch die fehlende PSM-Ebene in den MDA-Werkzeugen zusätzlich verstärkt. Nur eines der in dieser Arbeit evaluierten Tools verfügt standardmäßig über eine PSM-Ebene und diese auch nur für die J2EE Plattform. Die Vision der OMG, dass Softwaresysteme auf CIM-Ebene entworfen werden, ist in keinem evaluierten MDA-Werkzeug erkennbar.

Ob sich MDA in nächster Zeit als neues Programmierparadigma durchsetzen wird, liegt primär in der Hand der OMG und der MDA-Werkzeughersteller. Werden in naher Zukunft keine konkreten Standards für Transformationssprachen, ausdrucksstarke Modellierungsmöglichkeiten sowie einheitliche Profiles für die wichtigsten Softwareplattformen von der OMG veröffentlicht, könnte MDA einem ähnlichen Schicksal erliegen wie die CASE Tools vor einigen Dekaden. Zurzeit werden in den MDA-Werkzeugen nur proprietäre Konzepte der Werkzeughersteller eingesetzt. Ausnahmen stellen UML und XMI dar, wobei bei XMI unterschiedliche Dialekte eingesetzt werden und so der Modellaustausch zwischen MDA-Werkzeugen nicht immer funktioniert.

Die OMG verspricht durch den Einsatz des MDA-Paradigmas enorme Vorteile, die nur bedingt durch heutige MDA-Werkzeuge erfüllt werden. Diese Erkenntnis sollte von der OMG stärker kommuniziert werden, um keine unerfüllbaren Erwartungen bei den EntwicklerInnen zu wecken, da die MDA-Werkzeuge heute noch keine Technologien darstellen, die auf Knopfdruck vollständig einsatzfähige Systeme produzieren.

6.2 Ausblick

MDA stellt ein relativ junges und weit reichendes Forschungsgebiet der Informatik dar und kann in der Praxis für sehr unterschiedliche Projekte eingesetzt werden. Daher wurden bei der Durchführung dieser Arbeit viele zusätzliche Fragen und Problemstellungen aufgeworfen, deren Klärung den Rahmen der Arbeit bei weitem

sprengen würde. Im Folgenden werden die wichtigsten offenen Problemstellungen und weiterführenden Aufgabenstellungen aufgelistet, die in dieser Arbeit nicht näher behandelt werden und als Startpunkt für weitere Arbeiten dienen können:

- Ein offener Punkt der Arbeit ist die Fragestellung, mit wie viel Aufwand eine auf .NET-basierende Applikation aus dem vorhandenen Modell der Terminverwaltung abzuleiten wäre. ArcStyler bietet eine .NET-Cartridge, die auf demselben Komponentenmetamodell wie die WLS-Cartridge aufbaut. Somit wäre zu klären, ob auf Modellebene Änderungen vorgenommen werden müssen und welche Arbeiten auf Codeebene notwendig sind. Diese Aufgabenstellung ist ein gutes Beispiel, um festzustellen, mit welchem Aufwand die Nutzung neuer Plattformen für vorhandene Systeme verbunden wäre.

- In dieser Arbeit wurde ausschließlich die Generierung von Anwendungen aus Modellen betrachtet, jedoch stellt der umgekehrte Weg, d.h. die Generierung von Modellen aus vorhandenen Systemen (Reverse Engineering), eine weitere große Herausforderung dar. Die aus den vorhandenen Systemen erstellten Modelle können für die Erstellung neuer Anwendungen, welche auf anderen Plattformen basieren, genutzt werden. Hier wäre zu prüfen, ob und wie umfangreich Funktionen für Reverse Engineering in MDA-Werkzeugen implementiert sind.
  Der Kriterienkatalog beinhaltet nur Kriterien für die Erstellung von Anwendungen aus Modellen. Dieser könnte durch weitere Kriterien für Reverse Engineering ergänzt werden.

- Bei der Fallstudie wurde auf bereits vorhandene Transformationsdefinitionen zurückgegriffen. Hier wäre noch zu klären, mit welchem Aufwand die Erstellung eigener Transformationen verbunden ist. Da die evaluierten MDA-Werkzeuge unterschiedliche Transformationsdefinitionsmöglichkeiten anbieten, wäre es interessant, die Vor- und Nachteile der einzelnen Ansätze herauszuarbeiten.

- Weiters wurde nur eine relativ kleine Webapplikation für die Fallstudie implementiert. Dabei wäre aber noch zu klären, wie sich zusätzliche Aspekte (wie Mehrbenutzerzugriff, Austausch der Modelle zwischen unterschiedlichen Werkzeugen und die Modellierung von Verbindungen („Brücken") zwischen eigenständigen Anwendungen) bei der Erstellung einer großen und

komplexen Anwendung, die mehrere EntwicklerInnen miteinschließt, auswirken.
Zusätzlich könnten Fallstudien für unterschiedliche Applikationsarten und -architekturen (z.B. Web Services) durchgeführt werden, um zu eruieren, welche Projekte vom Einsatz des MDA-Paradigmas profitieren und welche nicht.

Anhang A

Modell der Terminverwaltung

A-1 ArcStyler Projektdatei

siehe CDROM/Kalendar.asprj

A-2 Modell als XMI-Datei (XMI Version 1.1)

siehe CDROM/Kalendar.xml

A-3 automatisch generierte Dokumentation der Terminverwaltung

siehe CDROM/dokumentation/index.html

Anhang B

Code der Terminverwaltung

B-1 generierte Codeteile des EJB-Systems

siehe CDROM/gen/wsl8_gen/

B-2 generierte Codeteile des Web-Systems

siehe CDROM/gen/webacc_gen/

B-3 lauffähige EJB-Komponente

siehe CDROM/magister/gen/components/libs/ejbsDefaultServerApplication/wls8/ejbsDefaultServerApplication.ear

B-4 lauffähige Web-Komponente

siehe CDROM/magister/magister/gen/components/libs/webapp/webapp.war

B-5 Installationsanleitung

siehe CDROM/readme.pdf

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Andr04] AndroMDA Team: Doc for 3.0M3-SNAPSHOT. http://www.andromda .org (14.12.04), 2004

[Arlo04] J. Arlow, I. Neustadt: Enterprise Patterns and MDA, Building better software with Archetype Patterns and UML. Addison-Wesley Verlag, 2004

[AT04a] Accelerated Technology: Nucleus BridgePoint Development Suite, Getting Started Guide. mitgeliefert in der Evaluationsversion der Nucleus BridgePointumgebung 6.1, beziehbar unter http://www.acceleratedtechnology.com (12.1.2005), 2004

[AT04b] Accelerated Technology: Object Action Language Manual. mitgeliefert in der Evaluationsversion der Nucleus BridgePointumgebung 6.1, beziehbar unter http://www.acceleratedtechnology.com (12.1.2005), 2004

[Bare02] L. Baresi, R. Heckel: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. http://www.elet.polimi.it/upload/baresi/papers/ICGT.pdf (1.2.2005), 2002

[Bene04] G. Beneken et. al.: Referenzarchitekturen und MDA. http://www4.in.tum.de/~seifert/publications/2004_beneken_mda-refarch.pdf (15.2.2005), 2004

[CC04a] Compuware Corporation: OptimalJ: How model-driven development enhances productivity. http://javacentral.compuware.com/members/login/click.shtml?/members/downloads/pdf/OJModelDrivenDev.pdf (1.2.2005), 2004

[CC04b] Compuware Corporation: OptimalJ: How transformation patterns transform UML models into high-quality J2EE applications.

http://javacentral.compuware.com/members/login/click.shtml?/members/downloads/pdf/OJPatterns.pdf (1.2.2005), 2000

[CC04c] Compuware Corporation: OptimalJ Architecture Edition OptimalJ 3.3. http://javacentral.compuware.com/members/optimalj/documentation/v3.3/UserGuideAE.pdf (1.2.2005), 2004

[Chri04] A. Christoph: Describing Horizontal Model Transformations with Graph Rewriting Rules. Proceedings of Model-Driven Architecture: Foundations and Applications 2005 (Seite 76-91), 2004

[Dijk70] E. Dijkstra, C. Hoare: Structured Programming. Academic Press, 1972

[Gamm96] E. Gamma, R. Helm, R. Johnson: Design Patterns. Addison-Wesley Verlag, 1996

[Gard03] T. Gardner et. al.: A review of OMG MOF 2.0 Query/Views/ Transformations Submissions and Recommendations towards the final Standard. http://www.zurich.ibm.com/pdf/ebizz/gardner-etal.pdf (1.2.2005), 2003

[Heck00] R. Heckel, G. Engels: Graph Transformation and Visual Modeling Techniques. http://wwwcs.upb.de/cs/ag-engels/Papers/2000/HeckelEATCS00.pdf (12.2.2005), 2000

[IO03a] Interactive Objects Software GmbH: ArcStyler Platform Guide for ArcStyler Version 4.0. http://www.arcstyler.de/as_support/docu/ArcStyler_Platform_Guide.pdf (12.2.2005), 2003

[IO03b] Interactive Objects Software GmbH: ArcStyler MDA-Cartridge Development and Extensibility Guide for ArcStyler Version 4.0. http://www.arcstyler.de/as_support/docu/extensibility_guide.pdf (12.2.2005), 2003

[IO03c] Interactive Objects Software GmbH: ArcStyler 4.0 Product Background Information. http://www.arcstyler.de/as_support/brochures/ArcStyler4_Product_Backgrounder_E.pdf (12.2.2005), 2003

[IO03d] Interactive Objects Software GmbH: ArcStyler MDA-Cartridge Guide for BEA Weblogic Server 8.1 for ArcStyler Version 4.0. http://www.arcstyler.de/as_support/docu/WLS_Guide.pdf (12.2.2005), 2003

[ITU03] International Telecommunication Union: ITU-T Recommendation Z.142 - Testing and Test Control Notation version 3 (TTCN-3): Graphical presentation format. http://www.itu.int/itudoc/itu-t/aap/sg17aap/history/z142/ (12.2.2005), 2003

[JCP02] Java Community Process JSR-000040: Java Metadata Interface API Specification 1.0 Final Release, http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html (1.2.2005), 2002

[Kapp03] G. Kappel, M. Hitz: UML@Work, Von der Analyse zur Realisierung, 2. Auflage, dpunkt.verlag, 2003

[KC01] Kennedy Carter: UML ASL Reference Guide, ASL Language Level 2.5. http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_06v2_5c.pdf (15.2.2005), 2001

[KC02a] Kennedy Carter: Supporting Model Driven Architecture with eXecutable UML. http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_80v2_2.pdf (15.2.2005), 2002

[KC02b] Kennedy Carter: Configurable Code Generation in MDA with ICCG. http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_27v3_0.pdf (15.2.2005), 2002

[KC04] Kennedy Carter: Model Driven Architecture and eXecutable UML. http://www.vmasc.odu.edu/wscc/pres/kennedy.ppt (2.3.2005), 2004

[Klep03] A. Kleppe, J. Warmer, W. Bast: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Verlag, 2003

[Lang03] B. Langlois, N. Farcet: THALES recommendations for the final OMG standard on Query/Views/Transformations. http://www.softmetaware.com/oopsla2003/langlois.pdf (1.2.2005), 2003

[Mars03] F. Marschall, P. Braun: Model Transformations for the MDA with BOTL. http://www4.in.tum.de/~marschal/pub/marschall_braun-mdafa03.pdf (Stand 1.2.2005), 2003

[Matu03] M. Matula: NetBeans Metadata Repository. http://mdr.netbeans.org/MDR-whitepaper.pdf (12.2.2005), 2003

[Mell02] S. J. Mellor, M. J. Balcer: Executable UML, A Foundation for the Model-Driven Architecture. Erste Auflage, Addison-Wesley Verlag, 2002

[Mell04] S. Mellor, K. Scott, A. Uhl: MDA Distilled. Addison-Wesley Verlag, 2004

[OMG95] Object Management Group: Object Management Architecture Guide. http://www.omg.org/cgi-bin/apps/doc?ab/97-05-05.pdf (1.2.2005), 1995

[OMG00] Object Management Group: UML Profile for CORBA Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/00-10-01.pdf (15.2.2005), 2000

[OMG01] Object Management Group: Model Driven Architecture (MDA). http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf (1.2.2005), 2001

[OMG02a] Object Management Group: Request for Proposal: MOF 2.0 Query /Views / Transformations RFP. http://www.omg.org/cgi-bin/apps/do_doc?ad/02-04-10.pdf (1.2.2005), 2002

[OMG02b] Object Management Group: XMLMetadata Interchange (XMI) Specification. http://www.omg.org/cgi-bin/apps/doc?formal/02-01-01.pdf (1.2.2005), 2002

[OMG02c] Object Management Group: Meta Object Facility (MOF) Specification, Version 1.4. http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf (1.2.2005), 2002

[OMG03a] Object Management Group: MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf (1.2.2005), 2003

[OMG03b] Object Management Group: Unified Modeling Language Specification Version 1.5. http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf (1.2.2005), 2003

[OMG03c] Object Management Group: Common Warehouse Metamodel (CWM) Specification, Version 1.1. http://www.omg.org/cgi-bin/apps/doc?formal/03-03-02.pdf (1.2.2005), 2003

[OMG03d] Object Management Group: Object Constraint Language (OCL) Specification, Version 1.1. http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf (1.2.2005), 2000

[OMG03e] Object Management Group: UML 2.0 Testing Profile Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-03.pdf (15.2.2005), 2003

[OMG03f] Object Management Group: UML 2.0 Superstructure Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf (15.2.2005), 2003

[OMG04a] Object Management Group: UML Profile for Enterprise Distributed Object Computing (EDOC). http://www.omg.org/technology/documents/formal/edoc.htm (15.2.2005), 2004

[OMG04b] Object Management Group: UML Profile and Interchange Models for Enterprise Application Integration (EAI) Specification. http://www.omg.org/cgi-bin/apps/doc?formal/04-03-26.pdf (15.2.2005), 2004

[OMG04c] Object Management Group: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. http://www.omg.org/cgi-bin/apps/doc?ptc/04-06-01.pdf (15.2.2005), 2004

[OMG04d] Object Management Group: Metamodel and UML Profile for Java and EJB Specification. http://www.omg.org/cgi-bin/apps/doc?formal/04-02-02.pdf (15.2.2005), 2004

[OMG05] Object Management Group: UML Profile for Schedulability, Performance, and Time Specification. http://www.omg.org/cgi-bin/apps/doc?formal/05-01-02.pdf (15.2.2005), 2005

[Rais04] C. Raistrick et. al.: Model Driven Architecture with Executable UML. Erste Auflage, Cambridge University Press, Cambridge, 2004

[Seli03] B. Selic: An Overview of UML 2.0. http://www.omg.org/news/meetings/workshops/MDA_2004_Manual/0-3_Tutorial1_Selic.pdf (12.2.2005), 2003

[Sims04] O. Sims, Enterprise MDA or How Enterprise Systems Will Be Built. http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Sims.pdf (Stand: 15.2.2005), 2004

[Soft01] SOFTEAM: MDA – When a major software industry trend meets our toolset, implemented since 1994. http://www.objecteering.com/pdf/whitepapers/us/mda.pdf (12.2.2005), 2001

[Soft04a] SOFTEAM: Objecteering/UML Profile Builder User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/UMLProfileBuilder.pdf (12.2.2005), 2004

[Soft04b] SOFTEAM: Objecteering/J Language User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/JLanguage.pdf (12.2.2005), 2004

[Soft04c] SOFTEAM: Objecteering/J Libraries User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/JLibraries.pdf (12.2.2005), 2004

[Soft04d] SOFTEAM: Objecteering/Metamodel User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/Metamodel.pdf (12.2.2005), 2004

[Soft04e] SOFTEAM: Objecteering/UML Modeler User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/UMLModeler.pdf (12.2.2005), 2004

[Soft04f] SOFTEAM: Objecteering/UML Document Template Editor User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/DocTemplateEditor.pdf (12.2.2005), 2004

[Soft04g] SOFTEAM: Objecteering/UML Administrating Objecteering Sites User Guide Version 5.3. http://www.objecteering.com/pdf/doc/us/Administration.pdf (12.2.2005), 2004

[Soft99] SOFTEAM: UML Profiles and the J language: Totally control your application development using UML. http://www.objecteering.com/pdf/whitepapers/us/uml_profiles.pdf (12.2.2005), 1999

[SUN03a] Sun Microsystems: Java 2 Platform Enterprise Edition Specification, Version 1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf (1.2.2005), 2003

[SUN03b] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.1. http://java.sun.com/products/ejb/docs.html (1.2.2005), 2003

[Will03] E. Willink: UMLX: A graphical transformation language for MDA. http://se2c.uni.lu/tiki/se2c-bib_download.php?id=799 (1.2.2005), 2003